ZetScript 2.1.0

Jordi Espada

Version 2.0, May 2024

Table of Contents

Chapter 1. Introduction

ZetScript is a script language with a syntax inspired in ECMAScript or Javascript but also it brings a easy way to bind parts of your C++ code. ZetScript provides a virtual machine so the execution is quite fast. Because ZetScript syntax is almost less or more equal to Javascript you can edit the code with any editor that supports Javascript syntax notation.

1.1. Compile

To install ZetScript, first download last source code from https://zetscript.org and compile the project with the following steps,

1. First we have to configure the project using cmake application.

```
cmake -B build
```

2. Second we have to compile the project. Using GNU tool chain is done through this command,

```
make -C build
```

After the compilation, the ZetScript library (.a) and command line interpreter utility (*zs*) will be placed at *build* directory.

Optionally, cmake configuration comes with the following options:

- *-DCMAKE_BUILD_TYPE={Release|Debug}* : Configures ZetScript project for *Release* or *Debug* (by default is *Release*).
- *-DTESTS:BOOL={true|false}* : Configures ZetScript to build or not tests (by default is *FALSE*).
- *-DSAMPLES:BOOL={true|false}* : Configures ZetScript to build or not samples (by default is *FALSE*).
- *-DBUILD_SHARED_LIBS:BOOL={true|false}* : Configures ZetScript to build as *SHARED* (by default is *STATIC*).

1.2. Hello World

Once ZetScript is compiled we present a quick sample of "HelloWorld" application,

1. Create a filename named helloworld.zs and type the following sentence

```
Console::outln("Hello world!");
```

2. Save the file and do the following at command line,

```
zs helloworld.zs
```

You should see the "Hello world" message at the command line.

Chapter 2. The language

2.1. Statments

ZetScript Language is based on a set of statements formed by values, operators, expressions, keywords, declarations, expressions, conditionals, integrators and functions. All statements are separated by semicolons at the end.

Example of four statements,

```
var op1,op2,res;
op1 = 5;
op2 = 6;
res = 5+6;
```

The last example can be executed within one line,

```
var op1,op2,res; op1=5; op2=6; res = 5+6;
```

2.2. Comments

ZetScript support block and line comments.

2.2.1. Block comment

```
/*
this is a block comment
*/
```

2.2.2. Line comment

```
// This is a line comment
```

2.3. Literals

ZetScript supports the following literals,

- Boolean
- Integer
- Float
- String

2.3.1. Boolean

Boolean literals is represented as *true* or *false*.

```
true // true value
false // false value
```

2.3.2. Integer

Integer literals are represented as integer with range from $-(2^{b-1})$ to $2^{b-1}-1$ where $b=32$ or $b=64$ it depending whether ZetScript is compiled for 32bits or 64bits. *Integer* values can be represented as *decimal* value, *hexadecimal*, *binary* or *character* format.

```
10; // decimal
0x1a; // hexadecimal
11010b; // binary
'b'; // character
```

Zetscript supports the following operators for *Integer* literals,

Operator	Expression	Description	Example
Negate	`-integer`	Negated value	`-10`
Bitwise complement	`~integer`	Invert all bits	`~011010b // (-27)`

2.3.3. Float

Float literals are represented as IEEE-754 floating point numbers in 32-bit or 64 bit it depending whether ZetScript is compiled for 32bits or 64bits. *Float* values can be represented in decimal form or scientific notation form.

```
1.2 // decimal form
2.0e-2 // cientific notation form
```

Zetscript supports the following operators for *Float* literals,

Operator	Expression	Description	Example
Negate	`-float`	Negated value	`-10.5`

2.3.4. String

String literal is represented as string within double quotes (").

```
"this is a string"
```

2.4. Variables and scopes

2.4.1. Variable

A variable is a kind of element that can hold a variable. A variable is declared with **var** keyword.

```
var i; // Declared variable
```

By default, any declared variable is set as undefined, that tells a variable was not properly initialized.

```
var i; // Declared variable without initialization. Its value is set as 'undefined'
```

A variable can be initialized through operator assignment (aka =).

```
var i=0; // Declared variable with initialization
```

It can also declare multiple variables with or without initialization separated by comma (,).

```
var i=10, j=5, k; // Multiple declaration with and without initialization
```

2.4.2. Constant

A constant is a kind of element where its value is immutable on its in lifetime. A constant is declared as **const** and is mandatory to be initialized.

```
const NUM_ITEMS=10; // constant 'NUM_ITEMS' with value 10.
```

2.4.3. Scope

In ZetScript we have two types of basic scopes.

- Global
- Local

Variables declared on the main script are global and the others declared within a block (i.e a an statement that starts with '{' and ends with '}'), are local.

```
// Declares *i* variable as global
var i;

// block starts here
{
    // Declare *j* variable as local (you can also access to i).
    var j;
}
// block ends here, so *j* variable doesn't exist anymore
```

2.5. Data types

2.5.1. Undefined

The type *Undefined* is defined as non initialized value. A variable is instanced as *Undefined* once a its assings *undefined* value or is not initialized.

```
var a; // 'a' is undefined as default
var b=undefined; // assigns undefined value
```

2.5.2. Null

The type *Null* is defined as empty or invalid value. A variable is instanced as *Null* once a its assings *null* value.

```
var i=null;
```

2.5.3. Integer

The type *Integer* represents an integer value from -(2b-1) to 2b-1-1 where b=32 or 64 it depending whether ZetScript is compiled for 32bits or 64bits. A variable is initialized as *Integer* once it assigns *decimal*, *hexadecimal*, *binary* or *character* value.

```
var a=10; // decimal
var b=0x1a; // hexadecimal
var c=01001b; // binary
var d='b'; //  character
```

Operators

ZetScript supports the following operators for *Integer* variables,

Operator	Expression	Description	Result
Negate	`-variable`	Negates its value	`var i=10;` `var j=-i; // j=-10`
Bitwise complement	`~variable`	Invert all bits	`var i=00000011b;` `var j=~i; // j=11111100b or -4 decimal`
PreIncrement	`++variable`	Increments FIRST and THEN evaluates	`var i=0;` `var j=++i; // j=1, i =1`
PostIncrement	`variable++`	Evaluates FIRST and THEN increments	`var i=0;` `var j=i++; // j=0; i=1`
Predecrement	`--variable`	Decrements FIRST and THEN evaluates	`var i=0;` `var j=--i; // j=-1; i=-1`
Postdecrement	`variable--`	Evaluates FIRST and THEN decrements	`var i=0;` `var j=i--;//j=0;i=-1;`

Static functions

Integer::parse()

Converts a given value to *Integer*.

Syntax

```
parse(_value)
```

Parameters

- *value : value to be converted as _Integer*

Returns

Integer as a result of converted value

Example

```
Console::outln(Integer::parse("10"))
Console::outln(Integer::parse(15.5))
```

Console output:

```
10
15
```

2.5.4. Float

The type *Float* is represented as IEEE-754 floating point numbers in 32-bit or 64 bit depending whether ZetScript is compiled for 32bits or 64bits. A variable is initialized as *Float* once it assigns a decimal or scientific value notation forms.

```
var a=1.2 // decimal form value
var b=2.0e-2 // cientific notation form value
```

Operators

ZetScript supports the following operators for *Float* variables,

Operator	Expression	Description	Result
Negate	`-variable`	Evaluates negated variable	`var i=10.5;` `var j=-i; // j=-10.5`
PreIncrement	`++variable`	Increments FIRST and THEN evaluates	`var i=0.5;` `var j=++i; // j=1.5, i =1.5`
PostIncrement	`variable++`	Evaluates FIRST and THEN increments	`var i=0.5;` `var j=i++; // j=0.5; i=1.5`
Predecrement	`--variable`	Decrements FIRST and THEN evaluates	`var i=0.5;` `var j=--i; // j=-0.5; i=-0.5`
Postdecrement	`variable--`	Evaluates FIRST and THEN decrements	`var i=0.5;` `var j=i--;//j=0.5;i=-0.5;`

Static functions

Float::parse()

Converts a given value to *Float*.

Syntax

```
Float::parse(_value)
```

Parameters

- *_value* : value to be converted as *Float*.

Returns

Float as a result of converted value.

Example

```
Console::outln(Float::parse("10.5"))
Console::outln(Float::parse(15))
```

Console output:

```
10.500000
15.000000
```

2.5.5. Boolean

The type *Boolean* it defines a variable represented as *true* or *false* value. A variable is initialized as *Boolean* once it assigns a boolean value.

```
var b=false;
```

2.5.6. String

The type *String* it defines a variable represented as a string value. A variable is initialized as *String* once it assigns a string or a sequence of chars between double quotes.

```
var s="this is a string";
```

Properties

String::length

Returns the length of the string.

Example

```
Console::outln("hello world".length)
```

Console output:

```
11
```

Member functions

String::insertAt()

Inserts string or character value into the string at defined position.

Syntax

```
String::insertAt(_position,_value)
```

Parameters

- *_position*: Position where to insert. The position ranges within 0 to the length-1 of the string.
- *_value*: String or character value.

Returns

None

Example

```
var s="hell wd";

Console::outln("s => '{0}'",s);
s.insertAt(4,'o')
Console::outln("s.insertAt(4,'o') => '{0}'",s)
s.insertAt(7,"orl")
Console::outln("s.insertAt(7,\"orl\") => '{0}'",s)
```

Console output:

```
s => 'hell wd'
s.insertAt(4,'o') => 'hello wd'
s.insertAt(7,"orl") => 'hello world'
```

String::eraseAt(_position)

Removes the value at defined position of the string.

Parameters

- *_position*: The position where there's the value to remove. The position ranges within 0 to the length-1 of the string.

Returns

None

Example

```
var s="helilo world";

Console::outln("s => '{0}'",s);
s.eraseAt(3);
Console::outln("s.eraseAt(3) => '{0}'",s)
```

Console output:

```
s => 'helilo world'
s.eraseAt(3) => 'hello world'
```

String::toUpperCase()

Returns a copy of the string converted to uppercase.

Syntax

```
String::toUpperCase()
```

Parameters

None

Returns

A copy of the string converted to uppercase.

Example

```
var s="Hello World";
Console::outln("s => '{0}'",s)
Console::outln("s.toUpperCase() => '{0}'",s.toUpperCase())
```

Console output:

```
s => 'Hello World'
s.toUpperCase() => 'HELLO WORLD'
```

String::toLowerCase()

Returns a copy of the string converted to lowercase.

Syntax

```
String::toLowerCase()
```

Parameters

None

Returns

```
A copy of the string converted to lowercase.
```

Example

```
var s="Hello World";
Console::outln("s => '{0}'",s.toLowerCase());
Console::outln("s.toLowerCase() => '{0}'",s.toLowerCase());
```

Console output:

```
s => 'hello world'
s.toLowerCase() => 'hello world'
```

String::clear()

Erases the contents of the string.

Syntax

```
String::clear()
```

Parameters

None

Returns

None

Example

```
var s="Hello World"
Console::outln("s => '{0}'",s)
s.clear()
Console::outln("s.clear() => '{0}'",s)
```

Console output:

```
s => 'Hello World'
s.clear() => ''
```

String::replace()

Creates a new string as a result of the copy of this which all occurrences of a string are replaced with another string.

Syntax

```
String::replace(_matching_string, _replacement_string)
```

Parameters

- *_matching_string*: String ocurrence to be replaced.
- *_replacement_string*: Replacement string.

Returns

A new *String* as a copy of this with *matching_string_ replaced by* remplacement_string_.

Example

```
var s="My blue car with blue door and blue wheel"
Console::outln("s => '{0}'",s)
Console::outln("s.replace(\"blue\",\"green\") => '{0}'",s.replace("blue","green"))
```

Console output:

```
s => 'My blue car with blue door and blue wheel'
s.replace("blue","green") => 'My green car with green door and green wheel'
```

String::split()

Splits string into array by a separator.

Syntax

```
String::split(_separator)
```

Parameters

- *_separator* : String or character as separator.

Returns

An *Array* of strings as a result of splitting the string by the *_separator*.

Example

```
var s="The quick brown fox jumps over the lazy dog.";
Console::outln("s => '{0}'",s)
Console::outln("s.split(' ') => {0}",s.split(' '))
Console::outln("s.split(\"jumps\") => {0}",s.split("jumps"))
```

Console output:

```
s => 'The quick brown fox jumps over the lazy dog.'
s.split(' ') => ["The","quick","brown","fox","jumps","over","the","lazy","dog."]
s.split("jumps") => ["The quick brown fox "," over the lazy dog."]
```

String::contains()

Finds out whether a string contains a substring or not.

Syntax

```
String::contains(_value)
```

Parameters

- *_value*: A substring to find.

Returns

A *Boolean* as *true* if substring exist or *false* if not.

Example

```
var s="The quick brown fox jumps over the lazy dog.";
Console::outln("s => '{0}'",s)
Console::outln("s.contains(\"fo\") => {0}",s.contains("fo"))
Console::outln("s.contains(\"foy\") => {0}",s.contains("foy"))
```

Console output:

```
s => 'The quick brown fox jumps over the lazy dog.'
s.contains("fo") => true
s.contains("foy") => false
```

String::indexOf()

Returns the position of the first occurrence of a sequence of characters in the string.

Syntax

```
String::indexOf(_search_value)
```

Parameters

- *_search_value* : A sequence of characters to search for.

Returns

An *Integer* value >=0 representing the index position of first occurrence found, or -1 if not found

Example

```
var s="The quick brown fox jumps over the lazy dog.";
Console::outln("s => '{0}'",s)
Console::outln("s.indexOf(\"fo\") => {0}",s.indexOf("fo"))
Console::outln("s.indexOf(\"foy\") => {0}",s.indexOf("foy"))
```

Console output:

```
s => 'The quick brown fox jumps over the lazy dog.'
s.indexOf("fo") => 16
s.indexOf("foy") => -1
```

String::startsWith()

Finds out if the string starts with the specified characters.

Syntax

```
String::startsWith(_value)
```

Parameters

- *value : A _String*, representing the characters to check for.

Returns

A *Boolean* as *true* if *value matches starting string, otherwise it returns _false*.

Example

```
var s = "Hello";
Console::outln("s => '{0}'", s)
Console::outln("s.startsWith(\"Hel\") => '{0}'", s.startsWith("Hel"));
Console::outln("s.startsWith(\"llo\") => '{0}'",s.startsWith("llo"));
Console::outln("s.startsWith(\"o\") => '{0}'",s.startsWith("o"));
```

Console output:

```
s => 'Hello'
s.startsWith("Hel") => 'true'
s.startsWith("llo") => 'false'
s.startsWith("o") => 'false'
```

String::endsWith()

Finds out if the string ends with the specified characters.

Syntax

```
String::endsWith(_value)
```

Parameters

- *value : A _String*, representing the characters to check for.

Returns

A *Boolean* as *true* if *value matches ending string, otherwise it returns _false*.

Example

```
var s = "Hello";
Console::outln("s.endsWith(\"Hel\") => '{0}'", s.endsWith("Hel"));
Console::outln("s.endsWith(\"llo\") => '{0}'",s.endsWith("llo"));
Console::outln("s.endsWith(\"o\") => '{0}'",s.endsWith("o"));
```

Console output:

```
s.endsWith("Hel") => 'false'
s.endsWith("llo") => 'true'
s.endsWith("o") => 'true'
```

String::substring()

Returns new string as a copy of this between a starting and ending index.

Syntax

```
String::substring(_start_index, _end_index)
```

Parameters

- *_start_index*: An *Integer* value as the first position to include in the returned substring.
- *_end_index* (optional): An *Integer* value as the end position to exclude. If the value is omitted it copies the rest of the string.

Returns

A copy of the string between *start_index_ and* end_index_.

Example

```
var s="hello world";
Console::outln("s.substring(0) => '{0}'",s.substring(0))
Console::outln("s.substring(3) => '{0}'",s.substring(3))
Console::outln("s.substring(2,3) => '{0}'",s.substring(2,3))
Console::outln("s.substring(3,-2) => '{0}'",s.substring(3,-2))
```

Console output:

```
s.substring(0) => 'hello world'
s.substring(3) => 'lo world'
s.substring(2,3) => 'll'
s.substring(3,-2) => 'lo worl'
```

String::append()

Extends the string by appending another string at the end.

Syntax

```
String::append(_string)
```

Parameters

- *_string* : A string.

Returns

None

Example

```
var s="Hell"
Console::outln("s => '{0}'",s);
s.append('o')
Console::outln("s.append('o') => '{0}'",s);
s.append(" World")
Console::outln("s.append(\" World\") => '{0}'",s);
```

Console output:

```
s => 'Hell'
s.append('o') => 'Hello'
s.append(" World") => 'Hello World'
```

Static functions

String::format()

Returns a formatted string.

Syntax

```
String::format(_string,..._args)
```

Parameters

- *_string* : A *String* representation with specified arguments and/or its format. Argument format is defined in the *_string* value within curly brackets (i.e between '{' and '}') where it specifies argument number and its format. The argument format is described below,

Format	description
{n}	It replaces {*n*} by the argument *n*
{n:dm}	It replaces the argument *n* padding *m* 0s on its left
{n,m}	It replaces argument *n* padding *m* spaces on its left

- *_args* : variable args to be replaced in the string representation

Returns

A new *String* as a result of formatting *string_ with* args_.

Example

```
Console::outln(String::format("Format first arg => {0}",1))
Console::outln(String::format("Format first arg => {0} and second arg => {1}",1,2))
Console::outln(String::format("Format first arg => {0} and another first arg => {0}",1))
Console::outln(String::format("Padding 2 0s => {0:d2}",1))
Console::outln(String::format("Padding 4 0s => {0:d4}",1))
Console::outln(String::format("Padding 2 spaces => {0,2}",1))
Console::outln(String::format("Padding 4 spaces => {0,4}",1))
```

Console output:

```
Format first arg => 1
Format first arg => 1 and second arg => 2
Format first arg => 1 and another first arg => 1
Padding 2 0s => 01
Padding 4 0s => 0001
Padding 2 spaces =>  1
Padding 4 spaces =>    1
```

2.5.7. Array

The type *Array* is a container that stores elements in a unidimensional array. A variable is instanced as *Array* once it assings a open/closed square brackets.

```
[]; // An instanced array
```

Variable instanced as *Array* can be initialized with a sequence of elements separated with coma.

```
[1,"string",true,2.0]; // An instanced array with elements
```

To access element array is done through *Integer* value as a index.

```
var v=[1,"string",true,2.0]; // variable 'v' has 4 elements where its access exist in [0..3]
v[1]; // It access array's second element (i.e "string")
```

Properties

Array::length

Returns the number of elements in the array.

Example

```
Console::outln([1,"string",true,2.0].length)
```

Console output:

```
4
```

Member functions

Array::push()

Adds a new element at the end of the array.

Syntax

```
Array::push(_element)
```

Parameters

- *_value* : Element to be added.

Returns

None

Example

```
var v=[0,1,2,3]
Console::outln("v => {0}",v)
v.push(4)
Console::outln("v.push(4) => {0}",v)
```

Console output:

```
v => [0,1,2,3]
v.push(4) => [0,1,2,3,4]
```

Array::pop()

Removes the last element in the array.

```
Array::pop()
```

Parameters

None

Returns

Last element of the array before be removed.

Example

```
var v=[0,1,2,3,4]
Console::outln("v => {0}",v)
var r=v.pop()
Console::outln("v.pop() => v:{0} r:{1}",v,r)
```

Console output:

```
v => [0,1,2,3,4]
v.pop() => v:[0,1,2,3] r:4
```

Array::insertAt()

Inserts an element in the array at defined position.

```
Array::insertAt(_position,_element)
```

Parameters

- *_position*: An *Integer* value as the position where to insert. The position ranges within 0 to the length-1 of the array.
- *_element*: Element to insert.

Example

```
var v=[0,2,3]
Console::outln("v => {0}",v)
v.insertAt(1,1)
Console::outln("v.eraseAt(1) => {0}",v)
```

Console output:

```
s => 'hell wd'
s.insertAt(4,'o') => 'hello wd'
s.insertAt(7,"orl") => 'hello world'
```

Array::eraseAt()

Removes an element at defined position.

```
Array::eraseAt(_position)
```

Parameters

- *_position*: An *Integer* as the position where there's the element to remove. The position ranges within 0 to the length-1 of the array.

Returns

None

Example

```
var v=[0,1,2,3]
Console::outln("v => {0}",v)
v.eraseAt(1)
Console::outln("v.eraseAt(1) => {0}",v)
```

Console output:

```
v => [0,1,2,3]
v.eraseAt(1) => [0,2,3]
```

Array::clear()

Clears all elements in the array.

```
Array::clear()
```

Parameters

None

Returns

None

Example

```
var v=[0,1,2,3]
Console::outln("v => {0}",v)
v.clear()
Console::outln("v => {0}",v)
```

Console output:

```
v => [0,1,2,3]
v => []
```

Array::join()

Returns a new string as the concatenation of all array elements with a separator.

```
Array::join(_separator)
```

Parameters

- *_separator* : A *String* or character as separator.

Returns

A string as a concatenation of all array elements with *_separator*.

Example

```
var v=["The","quick","brown","fox","jumps","over","the","lazy","dog."];
Console::outln("v => '{0}'",v)
Console::outln("v.join(' ') => {0}",v.join(' '))
```

Console output:

```
v => '["The","quick","brown","fox","jumps","over","the","lazy","dog."]'
v.join(' ') => The quick brown fox jumps over the lazy dog.
```

Array::contains()

Finds out if the array contains a element or not.

```
Array::contains(_element)
```

Parameters

- *_element*: An element to find.

Returns

A *Boolean* as *true* if the element exist or *false* if not.

Example

```
var v=[1,"string",false,10.5]
Console::outln("v => {0}",v)
Console::outln("v.contains(\"string\") => {0}",v.contains("string"))
Console::outln("v.contains(10) => {0}",v.contains(10))
Console::outln("v.contains(10.5) => {0}",v.contains(10.5))
```

Console output:

```
v => [1,"string",false,10.500000]
v.contains("string") => true
v.contains(10) => false
v.contains(10.5) => true
```

Array::extend()

Appends at the end of the array all elements from other array.

```
Array::extend(_array)
```

Parameters

- *_array* : An array.

Returns

None

Example

```
var v=[0,1,2,3]
Console::outln("v => {0}",v)
v.extend([4,5,6,7])
Console::outln("v.extend([4,5,6,7]) => {0}",v)
```

Console output:

```
v => [0,1,2,3]
v.extend([4,5,6,7]) => [0,1,2,3,4,5,6,7]
```

Static functions

Array::concat()

Creates a new array as a result of merging all elements of two arrays.

Syntax

```
Array::concat(_array1, _array2)
```

Parameters

- *_array1* : First array to merge.
- *_array2* : Second array to merge.

Returns

A new array as a result of merging all elements from *array1_ and* array2_.

Example

```
var v=[0,1,2,3]
Console::outln("v => {0}",v)
Console::outln("Array::concat(v,[4,5,6,7]) => {0}",Array::concat(v,[4,5,6,7]))
```

Console output:

```
v => [0,1,2,3]
Array::concat(v,[4,5,6,7]) => [0,1,2,3,4,5,6,7]
```

2.5.8. Object

The type *Object* is a container that stores a serie of key-value data. A variable is instanced as *Object* once it assings a open/closed of curly brackets.

```
{}; // An instanced object
```

Optionally it can be initialized with a series of pair key-value separated by comma.

```
// An instanced object with elements
{
  i:1
 ,s:"string"
 ,b:true
 ,f:2.0
};
```

The access to its fields is done through the variable object followed by '.' and key name or key name as string within brackets (i.e ["key_name"]),

```
var o={
  i:1
 ,s:"string"
 ,b:true
 ,f:2.0
};

o.i; // access field 'i' by '.'
o["i"]; // access field 'i' by '[]'
```

Static functions

Object::clear()

Clears all elements of an object.

Syntax

```
Object::clear(_object)
```

Parameters

- _object: The object to clear.

Returns

None

Example

```
var o={a:1,b:2,c:3,d:4};

Console::outln("o => {0}",o)
Object::clear(o)
Console::outln("Object::clear(o) => {0}",o)
```

Console output:

```
o => {"a":1,"b":2,"c":3,"d":4}
Object::clear(o) => {}
```

Object::erase()

Removes one element of an object.

Syntax

```
Object::erase(_object,_key_id)
```

Parameters

- *_object*: An object to erase the element.
- *_key_id*: A *String* value as the key id of the element to remove.

Returns

None

Example

```
var o={a:1,b:2,c:3,d:4};

Console::outln("o => {0}",o)
Object::erase(o,"b");
Console::outln("Object::erase(o,\"b\") => {0}",o)
```

Console output:

```
o => {"a":1,"b":2,"c":3,"d":4}
Object::erase(o,"b") => {"a":1,"c":3,"d":4}
```

Object::contains()

Finds out if the object contains a key or not.

Syntax

```
Object::contains(_object,_key_id)
```

Parameters

- *_object*: The object to check.
- *_key*: A *String* value as the key to find.

Returns

A *Boolean* as *true* if the key exist or *false* if not.

Example

```
var o={a:1,b:"string",c:false,d:10.5}

Console::outln("o => {0}",o)
Console::outln("Object::contains(o,\"b\") => {0}",Object::contains(o,"b"))
Console::outln("Object::contains(o,\"string\") => {0}",Object::contains(o,"string"))
```

Console output:

```
o => {"a":1,"b":"string","c":false,"d":10.500000}
Object::contains(o,"b") => true
Object::contains(o,"string") => false
```

Object::extend()

Appends at the end of an object all fields from other object.

Syntax

```
Object::extend(_dst,_src)
```

Parameters

- *_dst* : Destination object to extend.
- *_src* : Source object to copy all fields from.

Returns

None

Example

```
var o={a:1,b:2,c:3,d:4};
var e={e:5,f:6}

Console::outln("o => {0}",o)
Console::outln("e => {0}",e)
Object::extend(o,e)
Console::outln("Object::extend(o,e) => {0}",o)
```

Console output:

```
o => {"a":1,"b":2,"c":3,"d":4}
e => {"e":5,"f":6}
Object::extend(o,e) => {"a":1,"b":2,"c":3,"d":4,"e":5,"f":6}
```

Object::concat()

Creates a new object as a result of merging all fields of two objects.

Syntax

```
Object::concat(_object1, _object2)
```

Parameters

- *_object1* : First object to merge.
- *_object2* : Second object to merge.

Returns

A new object as a result of merging all fields from *object1_ and* object2_.

Example

```
var o={a:1,b:2,c:3,d:4};
var e={e:5,f:6}

Console::outln("o => {0}",o)
Console::outln("e => {0}",e)
Console::outln("Object::Concat(o,e) => {0}",Object::concat(o,e))
```

Console output:

```
o => {"a":1,"b":2,"c":3,"d":4}
e => {"e":5,"f":6}
Object::Concat(o,e) => {"a":1,"b":2,"c":3,"d":4,"e":5,"f":6}
```

Object::keys()

Returns an array of strings representing the key names of the field's object.

Syntax

```
Object::keys(_object)
```

Parameters

- *_object* : The object to get the keys.

Returns

An array of strings as key names of *_object*.

Example

```
var o={a:1,b:2,c:3,d:4};

Console::outln("o => {0}",o)
Console::outln("Object::keys(o) => {0}",Object::keys(o))
```

Console output:

```
o => {"a":1,"b":2,"c":3,"d":4}
Object::keys(o) => ["a","b","c","d"]
```

2.5.9. Function

The type *Function* contains function information to be invoqued at some place through paranthesis. A variable is instanced as *Function* once it assings a function.

Assing normal function to variable:

```
function add(_a,_b){
    return _a+_b
}

// function 'add' assigned to fun
var fun=add;

Console::outln(fun(5,5)); // Prints "10"
```

Also it can assing anonymous function to variable:

```
// anonymous function assigned to fun
var fun=function(_a,_b){
    return _a+_b
};

Console::outln(fun(5,5)); // it prints "10"
```

For more information about *Function* go to section 2.9.

2.5.10. Class

The type *Class* is a custom defined object that operates with its variables and functions. A *Class* is defined by keyword *class* followed by name.

```
// Definition class A
class A{
}
```

A variable is instanced as the class type through operator *new* followed by the class name defined with parethesis.

```
var a=new A(); // Instances object class type A
```

Also a class type can extend from another class with *extends* keyword.

```
// Definition class A
class A{

}

// Definition class B that extends from B
class B extends A{

}
```

For more information about *Class* go to section 2.10.

2.6. Operators

ZetScript supports the following operator types,

- Arithmetic operators
- Bitwise operators
- Assignment Operators
- Relational operators
- Logical operators
- Type operators

2.6.1. Arithmetic operators

Arithmetic operators performs arithmetic operations between two values. The arithmetic operators are the following,

Operator	Name	Description	Example

+	Add	It performs *addition* operation between two integer or float values or concatenates strings, arrays or objects	`1.5+6;// = 7.5`
			`"string_"+"string";// ="string_string"`
			`[1]+[2,3];// [1,2,3]`
			`{a:1}+{b:2,c:3}; //{a:1,b:2,c:3}`

-	Subtract	Performs a *substraction* operation between two integer/float values	`10-5; // = 5` `2.5-1;// = 1.5`
*	Multiply	Performs a *multiplication* between two integer/float values	`10*5; //= 50` `1.5*2;//= 3.0`
/	Divide	Performs a *division* between two integer/float values	`10/2; // = 5` `3/2.0 // = 1.5`

%	Modulus	Performs a *modulus* between two integer/float values	`3%2; //it results 1.0`
			`23.6%2; // it results 1.6`

2.6.2. Bitwise operators

Bitwise operators combines operations between two integer values. The bitwise operators are the following,

Operator	Name	Description	Example
&	Bitwise And	Performs *bitwise AND* operation between two integers	`0xa & 0x2; // = 0x2` `0xff & 0xf0; // = 0xf0`
|	Bitwise Or	Performs *bitwise OR* operation between two integers	`0xa | 0x5; // = 0xf` `0x1 | 0xe; // = 0xf`
^	Bitwise Xor	Performs *bitwise XOR* between two integers	`0xa ^ 0xa; // = 0x0` `0xa ^ 0x5; // = 0xf`
<<	Bitwise shift left	Performs *bitwise SHIFT LEFT* between two integers	`0x1 << 2; // = 0x4`
>>	Bitwise shift right	Performs *bitwise SHIFT RIGHT* between two integers	`0xff >> 1; // = 0x7f`
~	Bitwise complement	Performs *bitwise complement* operation from integer value	`~0xa; // = 0x7f`

2.6.3. Assigment operators

Assignment operators assign values to ZetScript variables. The assigment operators are the following,

Operator	Name	Description	Example

=	Assigment	It performs an assigment or multiple assigment	`var i,j;`
			`i=10;`
			`j=5;`
			
			`// multiple assigment`
			`i,j=j,i; // it does a swap ( i=5, j=10 )`

+=	Add assigment	Performs an *addition* to a variable by the value of the right operand and assigns the result to the variable. In case of string, array or object it performs a concatenation	`var i=1.5;`
			`i+=6; // i == 7.5`
			
			`var s="string";`
			`s+="_string";// s == "string_string"`
			
			`var v=[1];`
			`v+=[2,3];// v == [1,2,3]`
			
			`var o={a:1};`
			`o+={b:2,c:3}; // o == {a:1,b:2,c:3}`

-=	Substract assigment	Performs a *subtraction* to a variable by the value of the right operand and assigns the result to the variable	`var i=10;` `i-=5; // i == 5`
*=	Multiply assigment	Performs a *multiplication* to a variable by the value of the right operand and assigns the result to the variable	`var i=10` `i*=5; // i == 50`
/=	Divide assigment	Performs a *division* to a variable by the value of the right operand and assigns the result to the variable	`var i=10;` `i/=2; // i == 5.0`
%=	Modulus assigment	Performs *modulus* to a variable by the value of the right operand and assigns the result to the variable	`var i=50` `i%=100; // i == 50.0`
&=	Bitwise AND assigment	Performs *bitwise AND* to a variable by the value of the right operand and assigns the result to the variable	`var i=0xff;` `i&=0xf0; // i == 0xf0`
|=	Bitwise OR assigment	Performs *bitwise OR* to a variable by the value of the right operand and assigns the result to the variable	`var i=0xa` `i|=0x5; // i == 0xf`
^=	Bitwise XOR assigment	Performs *bitwise XOR* to a variable by the value of the right operand and assigns the result to the variable	`var i=0xa` `i^=0xa; // i == 0x0`
<<=	Bitwise SHIFT LEFT assigment	Performs _bitwise SHIFT LEFT to a variable by the value of the right operand and assigns the result to the variable	`var i=0x1;` `i<<=2; // i == 0x4`
>>=	Bitwise SHIFT RIGHT assigment	Performs bitwise SHIFT RIGHT to a variable by the value of the right operand and assigns the result to the variable	`var i=0xff;` `i>>=1; // i == 0x7f`

2.6.4. Relational operators

Relational operators evaluates relations through two values. The relational operators are the following,

Operator	Name	Description	Example
==	Equal	Check whether two values are equal	`10==10;// = true` `"hello"=="bye"; // = false`
!=	Not equal	Check whether two values are not equal	`10!=10; // = false` `"hello"!="bye"; // = true`
<	Less than	Checks whether first value is less than second value	`10<20; // = true` `20<10; // = false`
>	Greater than	Checks whether first value is greater than second value	`10>20; // = false` `20>10; // = true`
<=	Less equal than	Checks whether first value is less equal than second value	`10<=10; //= true` `11<=10; // = false`
>=	Greater equal than	Checks whether first value is greater equal than second value	`10>=11; // = false` `11>=10; // = true`

in	in	Performs the following operations:
		• Checks if a value exist in a string or array
		• Checks a key exist in object
			`'a' in "abc" // true`
			`'d' in "abc" // false`
			`"string" in [1,"string",false,10.5] // true`
			`10 in [1,"string",false,10.5] // false`
			`"a" in {a:1,b:2,c:3} // true`
			`"d" in {a:1,b:2,c:3} // false`

2.6.5. Logical operators

Logical operators combines operations between two boolean values. The logic operators are the following,

Operator	Name	Description	Example
&&	Logic And	Performs an AND operation between two boolean values	`true && true;// = true` `true && false;// = false`
||	Logic Or	Performs an OR operation between two Boolean values	`true || false;// = true` `false || false;// = false`

!	Logic Not	Negates boolean value	`!true; // = false`
			`!false; // = true`
			`!(true || false) // = false`
			`!(true && false) // = true`

2.6.6. Data Type operators

The instanceof operator

The *instanceof* operator returns *true* whether the object is an instance of the specified type (or class or subclass).

Example

```
class A{}
class B extends A{}

Console::outln("10 instanceof Integer => " + 10 instanceof Integer)
Console::outln("10 instanceof Float => " + 10 instanceof Float)
Console::outln("10.5 instanceof Integer => "+ 10.5 instanceof Integer)
Console::outln("10.5 instanceof Float => "+ 10.5 instanceof Float)
Console::outln("\"string\" instanceof String  =>  "+ "string" instanceof String)
Console::outln("[] instanceof Array  => "+ [] instanceof Array)
Console::outln("{} instanceof Object => "+ {} instanceof Object)
Console::outln("function(){} instanceof Function  => "+ function(){} instanceof Function)
Console::outln("new A() instanceof A => "+ new A() instanceof A)
Console::outln("new B() instanceof A => "+ new B() instanceof A)
Console::outln("new A() instanceof B => "+ new A() instanceof B)
```

Console output:

```
10 instanceof Integer => true
10 instanceof Float => false
10.5 instanceof Integer => false
10.5 instanceof Float => true
"string" instanceof String  =>  true
[] instanceof Array  => true
{} instanceof Object => true
function(){} instanceof Function  => true
new A() instanceof A => true
new B() instanceof A => true
new A() instanceof B => false
```

Typeof operator

The *typeof* operator returns the type of operand's value.

Example

```
class A{}

Console::outln("typeof 10 => " + typeof 10)
Console::outln("typeof 10.5 => "+ typeof 10.5)
Console::outln("typeof \"string\"  =>  "+ typeof "string")
Console::outln("typeof []  => "+ typeof [])
Console::outln("typeof {}  => "+ typeof {})
Console::outln("typeof function(){}  => "+ typeof function(){})
Console::outln("typeof new A() => "+ typeof new A())
```

Console output:

```
typeof 10 => type@Integer
typeof 10.5 => type@Float
typeof "string"  =>  type@String
typeof []  => type@Array
typeof {}  => type@Object
typeof function(){}  => type@Function
typeof new A() => type@A
```

2.6.7. Operator priority

Each operator it has priority in a evaluation. The priority of each operator is the following,

`in,instanceof,<<,>>,&,|,^,*,/,%,+,-,==,!=,>=,<=,>,<,&&,||`

Starting from left to right, the most left operator is the more priority and the most right one the less priority

For example this expression,

```
2+4*5; // will result 22
```

You can change the evaluation priority usign parenthesis.

For example,

```
(2+4)*5; // will result 36
```

2.7. Conditionals

2.7.1. If,else and else if statements

The if statement

The *if* statement is used to execute a block of code when a condition is *true*.

Syntax

```
if(condition){
    //Block of code to be executed if the condition is true
}
```

Example

```
if(value<10){
    Console::outln("value is less than 10")
}
```

The else statement

The *else* statement is used to execute a block of code when a condition is *false*.

Syntax

```
if(condition) {
    // do something if condition is true
}else{
    // do something if condition is false
}
```

Example

```
if(value<10){
    Console::outln("value is less than 10")
}else{
    Console::outln("value is greater equal than 10")
}
```

The else if statement

The *else if* statement is used to execute a block of code if first condition is *false*.

Syntax

```
if(condition1) {
    // do something if condition 1is true
}else if(condition2){
    // do something if condition 2 is true
}else{
    // do something if none of above conditions are true
}
```

Example

```
if(value>7){
    Console::outln("value greater than 7")
}else if(i>5){
    Console::outln("value greater than 5 and less equal than 7")
}else{
    Console::outln("value less equal than 5")
}
```

2.7.2. The ternary statement

The ternary statement to have a short *if-else* statement into single statement. It performs expression if the condition is *true* or the second expression if the condition is *false*.

Syntax

```
(condition)?expression_if_true:expression_if_false;
```

Example

```
var j = 0>1? 0:1; // j = 1
```

2.7.3. The switch statement

The *switch* statement is used to select one of many blocks of code to be executed.

Syntax

```
switch(expression) {
    case value_0:
        code block
        break;
    case value_1:
        code block
        break;
        ...
        case value_n
    default:
        code block
        break;
}
```

Example

```
switch(expr) {
    case 0:
    case 1:
        Console::outln("expr in [0,1]")
        break;
    case 2:
        Console::outln("expr is 2")
        break;
    default:
        Console::outln("expr is not in [0,1,2]")
        break;
}
```

2.8. Loops

2.8.1. The while loop

The *while* loop loops through a block of code as long as a specified condition is *true*.

Syntax

```
while(condition){
    // code block to be executed
}
```

Example

```
var i=0;

while(i < 5){
    Console::outln(i);
    i++;
}
```

Console output:

```
0
1
2
3
4
```

2.8.2. do-while loop

The *do-while* execute the code block once and then it will repeat the loop as long as the condition is *true*.

Syntax

```
do{
    // code block to be executed
} while (condition);
```

Example

```
var i=0;

do{
    Console::outln(i);
    i++;
}while(i < 5);
```

Console output:

```
0
1
2
3
4
```

2.8.3. The for loop

The *for* loop executes a block of code with 3 optional statements.

Syntax

```
for(statement1;statement2;statement3){
    // code block to be executed
}
```

- Statement 1 is executed before the loop (the code block) starts. Normally you will use statement 1 to initialize the variable used in the loop (for example var i = 0).
- Statement 2 defines the condition for running the loop.
- Statement 3 is executed each time after the code block has been executed.

Example

```
for(var i=0; i < 5; i++){
    Console::outln(i);
}
```

Console output:

```
0
1
2
3
4
```

2.8.4. The for-in loop

The *for-in* loop works as for loop but simplifies the iteration over containers like string, object or arrays. On each iteration it can evaluate its value in a variable or its key and its value in a pair of variables,

Syntax

```
// Evaluate container value per iteration
for(var v in array){
    // code block to be executed
}

// Evaluate container key and value per iteration
for(var k,v in array){
    // code block to be executed
}
```

Example

```
var object={a:10,b:10.5,c:"string",d:true}

Console::outln("Iterate object and get value\n")

for(var v in object){
    Console::outln("v => "+v);
}

Console::outln("\nIterate object and get pair key-value\n")

for(var k,v in object){
    Console::outln("k => "+k+" v => "+v);
}
```

Console output:

```
Iterate object and get value

v => 10
v => 10.500000
v => string
v => true

Iterate object and get pair key-value

k => a v => 10
k => b v => 10.500000
k => c v => string
k => d v => true
```

2.8.5. The break and continue

Break statement

The *break* statement is used to jump out of body loop.

Example

```
for(var i=0; i < 10; i++){

    // At i==4 breaks iteration
    if(i==4){
        break;
    }
    Console::outln(i);
}
```

Console output:

```
0
1
2
3
```

Continue statement

The *continue* statement breaks one iteration and continues with the next iteration in the loop.

Example

```
for(var i=0; i < 10; i++){
    if(i%2==1){
        continue;
    }

    // Only iterates within pair numbers
    Console::outln(i);
}
```

Console output:

```
0
2
4
6
8
```

2.9. Function

A function is a block of code to perform a particular task and is executed when in some part of the code it calls it

Syntax

```
function funName(arg1, arg2, ..., argn){
    // code to be executed
}
```

Example

```
function hello(_str){
    Console::outln("Hello "+str)
}
```

2.9.1. Function invocation

The call of a function is done when in some part of the code it calls it as follow,

Syntax

```
fun_name(arg1, arg2, arg3,..., argN);
```

Note: If a function is called with less than N args the rest of arguments will remain undefined as default.

Example

```
function hello(_str){
    Console::outln("Hello "+str)
}

hello("World")  // Prints Hello World
```

2.9.2. Return statement

The *return* statement ends function execution and specifies a value to be returned to the function caller.

Example

```
function add(op1, op2){
    return op1+op2;
}

add(4,5)  // returns 9
```

2.9.3. Function reference

A function can be stored in variables through its reference

Example

```
function add(op1, op2){
    return op1+op2;
}

var add_ref = add; // stored function add reference to fun_obj
var j=add_ref(2,3);// calls fun_obj (aka add) function. J=5
```

2.9.4. Anonymous functions

An *anonymous function* is a function without a name.

Syntax

```
function(arg1, arg2, ..., argN){
    // code to be executed.
};
```

An anonymous function is not accessible after its initial creation. Therefore, you often need to assign it to a variable.

Example

```
// Assigns anonymous function to 'add' variable
var add=function(op1, op2){
    return op1+op2;
};

// invokes function
var j=add(2,3); // j=5
```

2.9.5. Default parameters

Default function parameters allow named parameters to be initialized with default values if no value is passed.

Syntax

```
function fun(param_1 = default_value_1, /* ...,*/ param_n = default_value_N){
    // ...
};
```

Example

```
function sum(a, b = 1) {
  return a + b;
}

var j=sum(5); // j=6
```

2.9.6. Rest parameters

The rest parameters is an especial argument that allows pass indefinite number of arguments in an array. The rest parameter is declared by adding three dots (...) with the variable name at the end of the function argument list.

Syntax

```
function var_args(a,b,...args) {
    for(var arg in args){
        //
    }
}
```

Example

```
function number(str,...args) {
  var number = 0;
  for (var arg in args) {
        number += arg;
  }
  return str+number;
}

var res=number("Total number is = ",1, 10, 100, 1000, 10000);

Console::outln(res)
```

Console output:

```
Total number is = 11111
```

2.9.7. Parameter by reference

Parameter by reference allows write the argument taking its reference so the content will be modified outside the function. A parameter by reference is declared by prepending *ref* on argument.

Syntax

```
function(ref arg1, arg2, ..., ref arg_n){
    // code to be executed.
};
```

Example

```
function swap(ref _a, ref _b){
    var tmp=_a
    _a=_b;
    _b=tmp;
}

var i=5,j=10;

Console::outln("i:{0} j:{1}",i,j)

swap(i,j)

Console::outln("swap(i,j) => i:{0} j:{1}",i,j)
```

Console output:

```
i:5 j:10
swap(i,j) => i:10 j:5
```

2.10. Class

A class is a template for create objects with its own variables and functions. A class is defined using keyword *class* followed by the name.

```
class Test{


}
```

2.10.1. Class instantiantion

To instantiate an object of a defined class type, it is done using the *new* operator.

```
class Test{


}


var t= new Test(); // t instanced as class Test
```

2.10.2. Member functions

Member function

A function member aims to do operations or management with its class members. It's acceded through field operator ('.') and preceded by *this* keyword when is acceded within a member function. A member function can be implemented outside of the class too.

Example

```
class Test{

    // a member function
    method1(){
        Console::out("Access method 1")

        // call member function within a member function through 'this'
        this.method3();
    }

    // another member function
    method2(){
        Console::outln(" and method 3")
    }

};

// Test::method2 implementated outside of class Test
function Test::method2(){
    Console::outln("Access method 2")
}

var t=new Test()

t.method1(); // prints "Access method 1 and method 3"
t.method2(); // prints "Access method 2"
```

Static function

Static function is a kind of helper function of generic purposes about the class type. To declare static function it should be defined with static keyword. To access static function it has to type the name of containing class and '::' operator followed with the name of the function.

Example

```
class Test{

    // static function
    static staticMethod(){
        Console::outln("Access static method")
    }

};

Test::staticMethod(); // prints "Access static method"
```

2.10.3. Member variables

Member variables (also called fields) are data that live within the instanced class. To access a member variable is done through field operator ('.') and preceded by *this* keyword when is acceded within its member function

```
class Test{
    method3(){
        this.a=0;
    }
};


var t=new Test(); // t.a==undefined
t.method();       // t.a==0
t.a=10;           // t.a==10
```

In the code it can see that when *t* is instanced field 'a' is undefined. In the next section it'll shown some ways to initialize data when class is instantatiated.

Variable inicialization

Body

A class can initialize its variables defining them at its body with *var* and assignment operator (=)

```
class Test{

    // Declaration and initialization of member variable 'a' as 10
    var a=10;
}

var t=new Test(); // t.a==10
```

Constructor

A constructor is a special member function that is called when the class is instanced with the *new* operator. To create and initialize a variable member within *constructor* function is done through *this* keyword and member field operator ('.').

```
class Test{

    // Constructor function
    constructor(){
        this.a=10; // initialize member variable 'a' as 10
    }
}

var t=new Test(); // t.a==10
```

2.10.4. Inheritance

A class can inherit another class through *extends* keyword. The new extended class will copy all functions and variable initialization from base class. The extended class can call parent functions through *super* keyword.

```
class A {
    constructor(){
        this.a=10;
    }
};


class B extends A{
    constructor(){
        super(); // call A::constructor
        this.b=10 + this.a;
    }

};

var b=new B(); // b.a==10, b.b=20
```

2.10.5. Metamethods

Metamethods are special member functions that able to define operators seen on section 2.6. Metamethods can be static or member function depending whether the operation affects or not the object itself.

Member metamethods

_addassign()

Implements *addition assignment* operator (aka +=) with a value entered by parameter as right operand.

Syntax

```
_addassign(_value)
```

Parameters

- *_value* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _addassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ += _op1;
        }else if(_op1 instanceof Number){
            this.__value__ += _op1.__value__;
        }else{
            System::error("Number::_addassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(20);
Console::outln("number+=20 => {0}",number+=20)
Console::outln("number+=new Number(30) => {0}",number+=new Number(30))
```

Console output:

```
number+=20 => 40
number+=new Number(30) => 70
```

_andassign()

Implements *bitwise AND assignment* operator (aka &=) with a value entered by parameter as right operand.

Syntax

```
_andassign(_value)
```

Parameters

- *_value* : Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _andassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ &= Integer::parse(_op1);
        }else if(_op1 instanceof Number){
            this.__value__ &= Integer::parse(_op1.__value__);
        }else{
            System::error("Number::_andassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(0xff);
Console::outln("number&=0x3 => {0}",number&=0x3)
Console::outln("number&=new Number(0x1) => {0}",number&=new Number(0x1))
```

Console output:

```
number&=0x3 => 3
number&=new Number(0x1) => 1
```

_divassign()

Implements *division assignment* operator (aka /=) with a value entered by parameter as right operand.

Syntax

```
_divassign(_value)
```

Parameters

- *_value* : Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _divassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ /= _op1;
        }else if(_op1 instanceof Number){
            this.__value__ /= _op1.__value__;
        }else{
            System::error("Number::_divassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(20);
Console::outln("number/=20 => {0}",number/=20)
Console::outln("number/=new Number(30) => {0}",number/=new Number(30))
```

Console output:

```
number/=20 => 1.000000
number/=new Number(30) => 0.033333
```

_in()

Implements *in* operator

Syntax

```
_in(_value)
```

Parameters

- *_value* : Value or variable as value to check whether exist or not in the containing class.

Returns

Boolean telling whether the _value exist in or not.

Example

```
class Data{
    constructor(){
        this.data=[0,1,1,10,3,4,6]
    }

    _in(_value){
        return _value in this.data;
    }
};

var data=new Data();

if(10 in data){
    Console::outln("10 is content in data")
}
```

Console output:

```
10 is content in data
```

_modassign()

Implements *modulus assignment* operator (aka %=) with a value entered by parameter as right operand.

Syntax

```
_modassign(_value)
```

Parameters

- *_value* : Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _modassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ %= _op1;
        }else if(_op1 instanceof Number){
            this.__value__ %= _op1.__value__;
        }else{
            System::error("Number::_modassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(250);
Console::outln("number%=30 => {0}",number%=30)
Console::outln("number%=new Number(100) => {0}",number%=new Number(100))
```

Console output:

```
number%=30 => 10.000000
number%=new Number(100) => 10.000000
```

_mulassign()

Implements *multiplication assignment* operator (aka *=) with a value entered by parameter as right operand.

Syntax

```
_mulassign(_value)
```

Parameters

- *_value* : Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _mulassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ *= _op1;
        }else if(_op1 instanceof Number){
            this.__value__ *= _op1.__value__;
        }else{
            System::error("Number::_mulassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(20);
Console::outln("number*=20 => {0}",number*=20)
Console::outln("number*=new Number(30) => {0}",number*=new Number(30))
```

Console output:

```
number*=20 => 400
number*=new Number(30) => 12000
```

_neg()

Implements *negate* pre operator (aka -a).

Syntax

```
_neg()
```

Parameters

None

Returns

- A new object as result of negate operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _neg(){
        return -this.__value__;
    }

};

var number=new Number(20)

Console::outln("-number => "+ (-number))
```

Console output:

```
-number => -20
```

_not()

Implements *not* pre operator (aka !a).

Syntax

```
_not()
```

Parameters

None

Returns

- Boolean value as a result of not operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _not(){
        return this.__value__==0 || this.__value__==undefined || this.__value__ == null;
    }

};

var number=new Number()

if(!number){
    Console::outln("Number is empty")
}
```

Console output:

```
Number is empty
```

_orassign()

Implements *bitwise OR assignment* operator (aka `|=`) with a value entered by parameter as right operand.

Syntax

```
_andassign(_value)
```

Parameters

- *_value* : Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _orassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ |= Integer::parse(_op1);
        }else if(_op1 instanceof Number){
            this.__value__ |= Integer::parse(_op1.__value__);
        }else{
            System::error("Number::_orassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(0x1);
Console::outln("number|=0x2 => {0}",number|=0x2)
Console::outln("number|=new Number(0x4) => {0}",number|=new Number(0x4))
```

Console output:

```
number|=0x2 => 3
number|=new Number(0x4) => 7
```

_postdec()

Implements *post decrement* operator (aka a--).

Syntax

```
_postdec()
```

Parameters

None

Returns

A new object with a value before perform post decrement operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _postdec(){
        return new Number(this.__value__--);
    }

    _tostring(){
        return this.__value__;
    }
}

var number=new Number(20);
Console::outln("number-- => "+(number--))
Console::outln("number => "+(number))
```

Console output:

```
number-- => 20
number => 19
```

_postinc()

Implements *post increment* operator (aka a++).

Syntax

```
_postinc()
```

Parameters

None

Returns

A new object with a value before perform post increment operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _postinc(){
        return new Number(this.__value__++);
    }

    _tostring(){
        return this.__value__;
    }
}

var number=new Number(20);
Console::outln("number++ => "+(number++))
Console::outln("number => "+(number))
```

Console output:

```
number++ => 20
number => 21
```

_predec()

Implements *pre decrement* operator (aka --a).

Syntax

```
_predec()
```

Parameters

None

Returns

A new object with a value before perform pre decrement operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _predec(){
        return new Number(--this.__value__);
    }

    _tostring(){
        return this.__value__;
    }
}

var number=new Number(20);
Console::outln("--number => "+(--number))
Console::outln("number => "+(number))
```

Console output:

```
--number => 19
number => 19
```

_preinc()

Implements *pre increment* operator (aka ++a).

Syntax

```
_preinc()
```

Parameters

None

Returns

A new object with a value before perform pre increment operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _preinc(){
        return new Number(++this.__value__);
    }

    _tostring(){
        return this.__value__;
    }
}

var number=new Number(20);
Console::outln("++number => "+(++number))
Console::outln("number => "+(number))
```

Console output:

```
++number => 21
number => 21
```

_set()

Implements *assignment* operator (aka =) with a value entered by parameter as right operand.

Syntax

```
_set(_value)
```

Parameters

- *_value* :Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }
    _set(_value){
        if(_value instanceof Integer || _value instanceof Float){
            this.__value__ = _value;
        }else if(_value instanceof Number){
            this.__value__ = _value.__value__;
        }else{
            System::error("Number::set : parameter not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var n=new Number ();
Console::outln("n => "+n)

n=10;
Console::outln("n=10 => "+n)

n=new Number(20);
Console::outln("n=new Number(20) => "+n)
```

Console output:

```
n => 0
n=10 => 10
n=new Number(20) => 20
```

_shlassign()

Implements *bitwise SHIFT LEFT assignment* operator (aka <<=) with a value entered by parameter as right operand.

Syntax

```
_shrassign(_value)
```

Parameters

- *_value* : Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _shlassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ <<= Integer::parse(_op1);
        }else if(_op1 instanceof Number){
            this.__value__ <<= Integer::parse(_op1.__value__);
        }else{
            System::error("Number::_shlassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(0x1);
Console::outln("number<<=1 => {0}",number<<=1)
Console::outln("number<<=new Number(1) => {0}",number<<=new Number(1))
```

Console output:

```
number<<=1 => 2
number<<=new Number(1) => 4
```

_shrassign()

Implements *bitwise SHIFT RIGHT assignment* operator (aka >>=) with a value entered by parameter as right operand.

Syntax

```
_shrassign(_value)
```

Parameters

- *_op1* : Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _shrassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ >>= Integer::parse(_op1);
        }else if(_op1 instanceof Number){
            this.__value__ >>= Integer::parse(_op1.__value__);
        }else{
            System::error("Number::_shrassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(0x10);
Console::outln("number>>=1 => {0}",number>>=1)
Console::outln("number>>=new Number(1) => {0}",number>>=new Number(1))
```

Console output:

```
number>>=1 => 8
number>>=new Number(1) => 4
```

_subassign()

Implements *substraction assignment* operator (aka -=) with a value entered by parameter as right operand.

Syntax

```
_subassign(_value)
```

Parameters

- *_value* : Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _subassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ -= _op1;
        }else if(_op1 instanceof Number){
            this.__value__ -= _op1.__value__;
        }else{
            System::error("Number::_subassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(20);
Console::outln("number-=20 => {0}",number-=20)
Console::outln("number-=new Number(30) => {0}",number-=new Number(30))
```

Console output:

```
number-=20 => 0
number-=new Number(30) => -30
```

_tostring()

Returns custom string when string operation operation is invoved.

Syntax

```
_tostring()
```

Parameters

None

Returns

A string as a result when string operation operation is invoved.

Example

```
class NumberWithoutString{
    constructor(_value=0){
        this.__value__=_value;
    }
};

class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _tostring(){
        return this.__value__; // returns this.__value__ when string operation is involved
    }
};

Console::outln("without _tostring => "+new NumberWithoutString(10))
Console::outln("with _tostring => "+new Number(10))
```

Console output:

```
without _tostring => {"__value__":10}
with _tostring => 10
```

_xorassign()

Implements *bitwise XOR assignment* operator (aka ^=) with a value entered by parameter as right operand.

Syntax

```
_xorassign(_value)
```

Parameters

- *_op1* : Value or variable as right operand.

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    _xorassign(_op1){
        if(_op1 instanceof Integer || _op1 instanceof Float){
            this.__value__ ^= Integer::parse(_op1);
        }else if(_op1 instanceof Number){
            this.__value__ ^= Integer::parse(_op1.__value__);
        }else{
            System::error("Number::_xorassign : right operand not supported");
        }
    }
    _tostring(){
        return this.__value__;
    }
};

var number=new Number(0);
Console::outln("number^=0xa => {0}",number^=0xa)
Console::outln("number^=new Number(0x9) => {0}",number^=new Number(0x9))
```

Console output:

```
number^=0xa => 10
number^=new Number(0x9) => 3
```

Static metamethods

_add()

Implements *add* operator (aka +) between first operand and second operand

Syntax

```
_add(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of add operation

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _add(_op1,_op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_add : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_add : left operand not supported");
        }

        return new Number(op1+op2);
    }
    _tostring(){
        return this.__value__;
    }
};

Console::outln("new Number(10) + new Number(20) => " +  (new Number(10) + new Number(20)));
Console::outln("new Number(10) + 20 => " + (new Number(10) + 20));
Console::outln("10 + new Number(20) => " + (10 + new Number(20)));
```

Console output:

```
new Number(10) + new Number(20) => 30
new Number(10) + 20 => 30
10 + new Number(20) => 30
```

_and()

Implements *bitwise AND* operator (aka &) between first operand and second operand

Syntax

```
_and(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise and operation

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _and(_op1,_op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = Integer::parse(_op1); // ensure that is integer
        }else if(_op1 instanceof Number){
            op1 = Integer::parse(_op1.__value__); // ensure that is integer
        }else{
            System::error("Number::_and : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = Integer::parse(_op2); // ensure that is integer
        }else if(_op2 instanceof Number){
            op2 = Integer::parse(_op2.__value__); // ensure that is integer
        }else{
            System::error("Number::_and : left operand not supported"); // op1 and op2 are integers and it can perform bitwise and with
integers
        }

        return new Number(op1&op2);
    }
    _tostring(){
        return this.__value__;
    }
};

Console::outln("new Number(0x7) & new Number(0x4) => " +  (new Number(0x7) & new Number(0x04)));
Console::outln("new Number(0x7) & 0x4 => " + (new Number(0x7) & 0x04));
Console::outln("0x7 & new Number(0x4) => " + (0x7 & new Number(0x04)));
```

Console output:

```
new Number(0x7) & new Number(0x4) => 4
new Number(0x7) & 0x4 => 4
0x7 & new Number(0x4) => 4
```

_div()

Implements *division* operator (aka /) between first operand and second operand

Syntax

```
_div(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of division operation

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _div(_op1,_op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_div : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_div : left operand not supported");
        }

        return new Number(op1/op2);
    }
    _tostring(){
        return this.__value__;
    }
};

Console::outln("new Number(10) / new Number(20) => " +  (new Number(10) / new Number(20)));
Console::outln("new Number(10) / 20 => " + (new Number(10) / 20));
Console::outln("10 / new Number(20) => " + (10 / new Number(20)));
```

Console output:

```
new Number(10) / new Number(20) => 0.500000
new Number(10) / 20 => 0.500000
10 / new Number(20) => 0.500000
```

_equ()

Implements *equal* operator (aka ==) between first operand and second operand

Syntax

```
_equ(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 and op2 are EQUAL
- False if op1 and op2 are NOT EQUAL

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _equ(_op1, _op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_equ : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_equ : left operand not supported");
        }
        return op1==op2;
    }
};

Console::outln("new Number(20) == 20 => "+(new Number(20) == 20))
Console::outln("20 == new Number(30) => "+(20 == new Number(30)))
Console::outln("new Number(30) == new Number(20) => "+(new Number(30) == new Number(20)))
```

Console output:

```
new Number(20) == 20 => true
20 == new Number(30) => false
new Number(30) == new Number(20) => false
```

_gt()

Implements *greater than* operator (aka >) between first operand and second operand

Syntax

```
_gt(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 is GREATHER THAN op2
- False if op1 is LESS OR EQUAL THAN op2

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _gt(_op1, _op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_gt : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_gt : left operand not supported");
        }
        return op1>op2;
    }
};

Console::outln("new Number(20) > 20 => "+(new Number(20) > 20))
Console::outln("20 > new Number(30) => "+(20 > new Number(30)))
Console::outln("new Number(30) > new Number(20) => "+(new Number(30) > new Number(20)))
```

Console output:

```
new Number(20) > 20 => false
20 > new Number(30) => false
new Number(30) > new Number(20) => true
```

_gte(_op1,_op2)

Implements *greater than or equal* operator (aka >=) between first operand and second operand

Syntax

```
_gte(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 is GREATHER THAN OR EQUAL op2
- False if op1 is LESS THAN op2

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _gte(_op1, _op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_gte : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_gte : left operand not supported");
        }
        return op1>=op2;
    }
};

Console::outln("new Number(20) >= 20 => "+(new Number(20) >= 20))
Console::outln("20 >= new Number(30) => "+(20 >= new Number(30)))
Console::outln("new Number(30) >= new Number(20) => "+(new Number(30) >= new Number(20)))
```

Console output:

```
new Number(20) >= 20 => true
20 >= new Number(30) => false
new Number(30) >= new Number(20) => true
```

_lt()

Implements *less than* operator (aka <) between first operand and second operand

Syntax

```
_lt(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 is LESS THAN op2
- False if op1 is GRATHER EQUAL THAN op2

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _lt(_op1, _op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_lt : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_lt : left operand not supported");
        }
        return op1<op2;
    }
};

Console::outln("new Number(20) < 20 => "+(new Number(20) < 20))
Console::outln("20 < new Number(30) => "+(20 < new Number(30)))
Console::outln("new Number(30) < new Number(20) => "+(new Number(30) < new Number(20)))
```

Console output:

```
new Number(20) < 20 => false
20 < new Number(30) => true
new Number(30) < new Number(20) => false
```

_lte()

Implements *less than or equal* operator (aka <=) between first operand and second operand

Syntax

```
_lte(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 is LESS THAN OR EQUAL op2
- False if op1 is GRATHER THAN op2

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _lte(_op1, _op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_lte : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_lte : left operand not supported");
        }
        return op1<=op2;
    }
};

Console::outln("new Number(20) <=  20 => "+(new Number(20) <= 20))
Console::outln("20 <= new Number(30) => "+(20 <= new Number(30)))
Console::outln("new Number(30) <= new Number(20) => "+(new Number(30) <= new Number(20)))
```

Console output:

```
new Number(20) <=  20 => true
20 <= new Number(30) => true
new Number(30) <= new Number(20) => false
```

_mod()

Implements *modulus* operator (aka *) between first operand and second operand

Syntax

```
_mod(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of modulus operation

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _mod(_op1,_op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_mod : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_mod : left operand not supported");
        }

        return new Number(op1%op2);
    }
    _tostring(){
        return this.__value__;
    }
};

Console::outln("new Number(10) % new Number(20) => " +  (new Number(10) % new Number(20)));
Console::outln("new Number(10) % 20 => " + (new Number(10) % 20));
Console::outln("10 % new Number(20) => " + (10 % new Number(20)));
```

Console output:

```
new Number(10) % new Number(20) => 10.000000
new Number(10) % 20 => 10.000000
10 % new Number(20) => 10.000000
```

_mul()

Implements *multiplication* operator (aka *) between first operand and second operand

Syntax

```
_mul(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of multiplication operation

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _mul(_op1,_op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_mul : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_mul : left operand not supported");
        }

        return new Number(op1*op2);
    }
    _tostring(){
        return this.__value__;
    }
};

Console::outln("new Number(10) * new Number(20) => " +  (new Number(10) * new Number(20)));
Console::outln("new Number(10) * 20 => " + (new Number(10) * 20));
Console::outln("10 * new Number(20) => " + (10 * new Number(20)));
```

Console output:

```
new Number(10) * new Number(20) => 200
new Number(10) * 20 => 200
10 * new Number(20) => 200
```

_nequ()

Implements *not equal* operator (aka !=) between first operand and second operand

Syntax

```
_nequ(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 and op2 are NOT EQUAL
- False if op1 and op2 are EQUAL

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _nequ(_op1, _op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = _op1;
        }else if(_op1 instanceof Number){
            op1 = _op1.__value__;
        }else{
            System::error("Number::_nequ : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = _op2;
        }else if(_op2 instanceof Number){
            op2 = _op2.__value__;
        }else{
            System::error("Number::_nequ : left operand not supported");
        }
        return op1!=op2;
    }
};

Console::outln("new Number(20) != 20 => "+(new Number(20) != 20))
Console::outln("20 != new Number(30) => "+(20 != new Number(30)))
Console::outln("new Number(30) != new Number(20) => "+(new Number(30) != new Number(20)))
```

Console output:

```
new Number(20) != 20 => false
20 != new Number(30) => true
new Number(30) != new Number(20) => true
```

_or()

Implements *bitwise OR* operator (aka |) between first operand and second operand

Syntax

```
_or(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise or operation

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _or(_op1,_op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = Integer::parse(_op1); // ensure that is integer
        }else if(_op1 instanceof Number){
            op1 = Integer::parse(_op1.__value__); // ensure that is integer
        }else{
            System::error("Number::_or : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = Integer::parse(_op2); // ensure that is integer
        }else if(_op2 instanceof Number){
            op2 = Integer::parse(_op2.__value__); // ensure that is integer
        }else{
            System::error("Number::_or : left operand not supported");
        }

        return new Number(op1|op2); // op1 and op2 are integers and it can perform bitwise or with integers
    }
    _tostring(){
        return this.__value__;
    }
};

Console::outln("new Number(0x1) | new Number(0x2) => " +  (new Number(0x1) | new Number(0x2)));
Console::outln("new Number(0x1) | 0x2 => " + (new Number(0x1) | 0x2));
Console::outln("0x1 | new Number(0x2) => " + (0x1 | new Number(0x2)));
```

Console output:

```
new Number(0x1) | new Number(0x2) => 3
new Number(0x1) | 0x2 => 3
0x1 | new Number(0x2) => 3
```

_shl()

Implements *bitwise SHIFT LEFT* operator (aka <<) between first operand and second operand

Syntax

```
_shl(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise shift left operation

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _shl(_op1,_op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = Integer::parse(_op1);// ensure that is integer
        }else if(_op1 instanceof Number){
            op1 = Integer::parse(_op1.__value__);// ensure that is integer
        }else{
            System::error("Number::_shl : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = Integer::parse(_op2);// ensure that is integer
        }else if(_op2 instanceof Number){
            op2 =Integer::parse(_op2.__value__);// ensure that is integer
        }else{
            System::error("Number::_shl : left operand not supported");
        }

        return new Number(op1<<op2); // op1 and op2 are integers and it can perform bitwise shift left with integers
    }
    _tostring(){
        return this.__value__;
    }
};

Console::outln("new Number(0x1) << new Number(2) => " +  (new Number(0x1) << new Number(2)));
Console::outln("new Number(0x1) << 2 => " + (new Number(0x1) << 2));
Console::outln("0x1 << new Number(2) => " + (0x1 << new Number(2)));
```

Console output:

```
new Number(0x1) << new Number(2) => 4
new Number(0x1) << 2 => 4
0x1 << new Number(2) => 4
```

_shr()

Implements *bitwise SHIFT RIGHT* operator (aka <<) between first operand and second operand

Syntax

```
_shr(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise shift right operation

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _shr(_op1,_op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = Integer::parse(_op1); // ensure that is integer
        }else if(_op1 instanceof Number){
            op1 = Integer::parse(_op1.__value__); // ensure that is integer
        }else{
            System::error("Number::_shr : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = Integer::parse(_op2); // ensure that is integer
        }else if(_op2 instanceof Number){
            op2 = Integer::parse(_op2.__value__) // ensure that is integer;
        }else{
            System::error("Number::_shr : left operand not supported");
        }

        return new Number(op1>>op2); // op1 and op2 are integers and it can perform bitwise shift right with integers
    }
    _tostring(){
        return this.__value__;
    }
};

Console::outln("new Number(0x8) >> new Number(2) => " +  (new Number(0x8) >> new Number(2)));
Console::outln("new Number(0x8) >> 2 => " + (new Number(0x8) >> 2));
Console::outln("0x8 >> new Number(2) => " + (0x8 >> new Number(2)));
```

Console output:

```
new Number(0x8) >> new Number(2) => 2
new Number(0x8) >> 2 => 2
0x8 >> new Number(2) => 2
```

_xor()

Implements *bitwise XOR* operator (aka ^) between first operand and second operand

Syntax

```
_xor(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise xor operation

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    static _xor(_op1,_op2){
        var op1,op2
        if(_op1 instanceof Integer || _op1 instanceof Float){
            op1 = Integer::parse(_op1);// ensure that is integer
        }else if(_op1 instanceof Number){
            op1 = Integer::parse(_op1.__value__);// ensure that is integer
        }else{
            System::error("Number::_xor : right operand not supported");
        }

        if(_op2 instanceof Integer || _op2 instanceof Float){
            op2 = Integer::parse(_op2);// ensure that is integer
        }else if(_op2 instanceof Number){
            op2 = Integer::parse(_op2.__value__);// ensure that is integer
        }else{
            System::error("Number::_xor : left operand not supported");
        }

        return new Number(op1^op2);// op1 and op2 are integers and it can perform bitwise xor with integers
    }
    _tostring(){
        return this.__value__;
    }
};

Console::outln("new Number(0xa) ^ new Number(0x9) => " +  (new Number(0xa) ^ new Number(0x9)));
Console::outln("new Number(0xa) ^ 0x9 => " + (new Number(0xa) ^ 0x9));
Console::outln("0xa ^ new Number(0x9) => " + (0xa ^ new Number(0x9)));
```

Console output:

```
new Number(0xa) ^ new Number(0x9) => 3
new Number(0xa) ^ 0x9 => 3
0xa ^ new Number(0x9) => 3
```

2.10.6. Properties

A property is a kind of variable that are acceded through a metamethods.

Syntax

```
class Test{

    // property
    property{
            ...
    }

}
```

Member functions

_get()

_get returns the value of the property

Syntax

```
_get()
```

Parameters

None

Returns

Returns the value of the property

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }
    }
}

var number=new Number(20);
Console::outln("number.value => "+number.value)
```

Console output:

```
number.value => 20
```

_set()

Implements *assignment* operator (aka =) with a value entered by parameter as right operand.

Syntax

```
_set(_value)
```

Parameters

- *_value* :Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.value=_value // calls Number::value::_set metamethod
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _set(_value){
            if(_value instanceof Integer || _value instanceof Float){
                this.__value__ = _value;
            }else if(_value instanceof Number){
                this.__value__ = _value.__value__;
            }else{
                System::error("Number::value::_set : parameter not supported");
            }
        }
    }
}

var number=new Number(20);
Console::outln("On operation 'number=new Number(20)' then 'number.value' is => "+number.value)

number.value = 10;
Console::outln("On operation 'number=10' the 'number.value' is => "+number.value)
```

Console output:

```
On operation 'number=new Number(20)' then 'number.value' is => 20
On operation 'number=10' the 'number.value' is => 10
```

_addassign()

Implements *addition assignment* operator (aka +=) with a value entered by parameter as right operand.

Syntax

```
_addassign(_value)
```

Parameters

- *_value* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _addassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ += _op1;
            }else{
                System::error("Number::value::_addassign : right operand not supported");
            }
        }
    }
}

var number=new Number(20);
Console::outln("number.value+=20 => {0}",number.value+=20)
```

Console output:

```
number.value+=20 => 40
```

_subassign()

Implements *substraction assignment* operator (aka -=) with a value entered by parameter as right operand.

Syntax

```
_subassign(_value)
```

Parameters

- *_value* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _subassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ -= _op1;
            }else{
                System::error("Number::value::_subassign : right operand not supported");
            }
        }
    }
}

var number=new Number(30);
Console::outln("number.value-=20 => {0}",number.value-=20)
```

Console output:

```
number.value-=20 => 10
```

_mulassign()

Implements *multiplication assignment* operator (aka *=) with a value entered by parameter as right operand.

Syntax

```
_mulassign(_value)
```

Parameters

- *_value* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _mulassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ *= _op1;
            }else{
                System::error("Number::value::_mulassign : right operand not supported");
            }
        }
    }
}

var number=new Number(10);
Console::outln("number.value*=10 => {0}",number.value*=10)
```

Console output:

```
number.value*=10 => 100
```

_divassign()

Implements *division assignment* operator (aka /=) with a value entered by parameter as right operand.

Syntax

```
_divassign(_value)
```

Parameters

- *_value* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _divassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ /= _op1;
            }else{
                System::error("Number::value::_divassign : right operand not supported");
            }
        }
    }
}

var number=new Number(10);
Console::outln("number.value/=20 => {0}",number.value/=20)
```

Console output:

```
number.value/=20 => 0.500000
```

_modassign()

Implements *modulus assignment* operator (aka %=) with a value entered by parameter as right operand.

Syntax

```
_modassign(_value)
```

Parameters

- *_value* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _modassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ %= _op1;
            }else{
                System::error("Number::value::_modassign : right operand not supported");
            }
        }
    }
}

var number=new Number(250);
Console::outln("number.value%=100 => {0}",number.value%=100)
```

Console output:

```
number.value%=100 => 50.000000
```

_andassign()

Implements *bitwise AND assignment* operator (aka &=) with a value entered by parameter as right operand.

Syntax

```
_andassign(_value)
```

Parameters

- *_value* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _andassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ &= Integer::parse(_op1);
            }else{
                System::error("Number::value::_andassign : right operand not supported");
            }
        }
    }
}

var number=new Number(0xf);
Console::outln("number.value&=0x3 => {0}",number.value&=0x3)
```

Console output:

```
number.value&=0x3 => 3
```

_orassign()

Implements *bitwise OR assignment* operator (aka |=) with a value entered by parameter as right operand.

Syntax

```
_orassign(_value)
```

Parameters

- *_value* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _orassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ |= Integer::parse(_op1);
            }else{
                System::error("Number::value::_orassign : right operand not supported");
            }
        }
    }
}

var number=new Number(0x1);
Console::outln("number.value|=0x2 => {0}",number.value|=0x2)
```

Console output:

```
number.value|=0x2 => 3
```

_xorassign()

Implements *bitwise XOR assignment* operator (aka ^=) with a value entered by parameter as right operand.

Syntax

```
_xorassign(_value)
```

Parameters

- *_op1* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _xorassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ ^= Integer::parse(_op1);
            }else{
                System::error("Number::value::_xorassign : right operand not supported");
            }
        }
    }
}

var number=new Number(0x0);
Console::outln("number.value^=0xa => {0}",number.value^=0xa)
```

Console output:

```
number.value^=0xa => 10
```

_shrassign()

Implements *bitwise SHIFT RIGHT assignment* operator (aka >>=) with a value entered by parameter as right operand.

Syntax

```
_shrassign(_value)
```

Parameters

- *_op1* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _shrassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ >>= Integer::parse(_op1);
            }else{
                System::error("Number::value::_shrassign : right operand not supported");
            }
        }
    }
}

var number = new Number(0x8);
Console::outln("number.value>>=1 => {0}",number.value>>=1)
```

Console output:

```
number.value>>=1 => 4
```

_shlassign()

Implements *bitwise SHIFT LEFT assignment* operator (aka <<=) with a value entered by parameter as right operand.

Syntax

```
_shlassign(_value)
```

Parameters

- *_value* : Value or variable as right operand

Returns

None.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _shlassign(_op1){
            if(_op1 instanceof Integer || _op1 instanceof Float){
                this.__value__ <<= Integer::parse(_op1);
            }else{
                System::error("Number::value::_shlassign : right operand not supported");
            }
        }
    }
}

var number=new Number(0x1);
Console::outln("number.value<<=1 => {0}",number.value<<=1)
```

Console output:

```
number.value<<=1 => 2
```

_postinc()

Implements *post_increment* operator (aka a++)

Syntax

```
_postinc()
```

Parameters

None

Returns

The value before perform post increment operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }


    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _postinc(){
            return this.__value__++;
        }
    }
}

var number=new Number(20);
Console::outln("number.value++ => {0}",number.value++)
Console::outln("number.value => {0}",number.value)
```

Console output:

```
number.value++ => 20
number.value => 21
```

_postdec()

Implements *post_decrement* operator (aka a--)

Syntax

```
_postdec()
```

Parameters

None

Returns

The value before perform post decrement operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _postdec(){
            return this.__value__--;
        }
    }
}

var number=new Number(20);
Console::outln("number.value-- => {0}",number.value--)
Console::outln("number.value => {0}",number.value)
```

Console output:

```
number.value-- => 20
number.value => 19
```

_preinc()

Implements *pre_increment* operator (aka ++a)

Syntax

```
_preinc()
```

Parameters

None

Returns

The value before perform pre increment operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }


    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _preinc(){
            return ++this.__value__;
        }
    }
}

var number=new Number(20);
Console::outln("++number.value => {0}",++number.value)
Console::outln("number.value => {0}",number.value)
```

Console output:

```
++number.value => 21
number.value => 21
```

_predec()

Implements *pre_decrement* operator (aka --a)

Syntax

```
_predec()
```

Parameters

None

Returns

The value before perform pre decrement operation.

Example

```
class Number{
    constructor(_value=0){
        this.__value__=_value;
    }

    // property 'value'
    value{
        _get(){
            return this.__value__
        }

        _predec(){
            return --this.__value__;
        }
    }
}

var number=new Number(20);
Console::outln("--number.value => {0}",--number.value)
Console::outln("number.value => {0}",number.value)
```

Console output:

```
--number.value => 19
number.value => 19
```

2.10.7. Iterator

An iterator is a type with a set of metamethods defined that allow iteration from a custom container type. The metamethods are the following,

- *_get()*: it returns a value according current index
- *_end()*: it returns a boolean value telling whether the iterator has finished or not
- *_next()*: it increments index iterator

Furthermore, it has to pass as a parameter the instance of the custom container type to reference its data. So, an iterator type should be defined as below,

```
class IteratorContainer{
    constructor(_container){
        this.container=_container;
        // ...
    }
    _get(){
        //...
        return value;
    }
    _end(){
        //...
        return [true or false]
    }
    _next(){
        // increments iterator
    }
}
```

Finally, in order to have a custom container type as iterable it has to implement *_iter* metamethod that returns the defined iterator passing the instance of current object (i.e this).

```
class Container{
    _iter(){
        return new IteratorContainer(this);
    }
}
```

Example

```
class ContainerIterator{
    constructor(_container){
        this.container=_container;
        this.index=0;
    }
    _get(){
        // returns 2 values: index as key and this.container.data[this.index] as value
        return this.index,this.container.data[this.index];
    }
    _next(){
        if(this._end()){
            return;
        }
        this.index++;

    }
    _end(){
        return this.index>=this.container.data.length;
    }
}

class Container{
    constructor(){
        this.data=[10,20,30,40,50,60,70]
    }

    _iter(){
        return new ContainerIterator(this)
    }
}

var container=new Container();
```

```
// iterate over all elements
for(var k,v in container){
    Console::outln("key: {0}  value: {1}",k,v)
}
```

Console output:

```
key: 0  value: 10
key: 1  value: 20
key: 2  value: 30
key: 3  value: 40
key: 4  value: 50
key: 5  value: 60
key: 6  value: 70
```

2.11. Standard Library

ZetScript includes a set of builtin modules in order to facilitate management in the script environment.

2.11.1. Console

Console module implements a set of functions for console.

Console::out()

Prints message to the console with a ending carry return.

```
Console::out(_value,..._args)
```

Parameters

- *_value* : value to print. If *_value* is string it can be formated (see in section String::format()), else it tries to print a the value as human readable.
- *_args* : variable args to be replaced if *_value* is string representation.

Returns

None

Example

```
Console::out("Hello")
Console::out(" ")
Console::out("World")
```

Console output:

```
Hello World
```

Console::outln()

Prints message to the console with a ending carry return.

Syntax

```
Console::outln(_value,..._args)
```

Parameters

- *_value*: value to print. If *_value* is string it can be formated (see in section String::format()), else it tries to print a the value as human readable.
- *_args*: variable args to be replaced if *_value* is string representation.

Returns

None

Example

```
Console::outln("Hello")
Console::outln()
Console::outln("World")
```

Console output:

```
Hello

World
```

Console::error()

Prints error message to the console.

```
Console::out(_value,..._args)
```

Parameters

- *_value*: value to print. If *_value* is string it can be formated (see in section String::format()), else it tries to print a the value as human readable.
- *_args*: variable args to be replaced if *_value* is string representation.

Returns

None

Console::errorln()

Prints error message to the console with a ending carry return.

Syntax

```
Console::outln(_value,..._args)
```

Parameters

- *_value*: Value to print. If *_value* is string it can be formated (see in section String::format()), else it tries to print a the value as human readable.
- *_args*: Variable args to be replaced if *_value* is string representation.

Returns

None

Console::readChar()

it waits untill reads a character from console.

Syntax

```
Console::readChar()
```

Parameters

None

Returns

The integer as the equivalent character readed from console.

Example

```
// waits until reads a char
var c=Console::readChar();

// prints readed char
Console::outln("Key pressed: {0}",c)
```

2.11.2. System

System module implements a set of functions for system.

System::clock()

Returns the amount of time in seconds.

Syntax

```
System::clock()
```

Parameters

None

Returns

The amount of time in seconds as *Float* type.

Example

```
while(System::clock() < 1){
    var start=System::clock()
    while((System::clock()-start) < 0.1){}
    Console::outln("Ellapsed seconds: {0}",System::clock());
}
```

Console output:

```
Ellapsed seconds: 0.101000
Ellapsed seconds: 0.202000
Ellapsed seconds: 0.303000
Ellapsed seconds: 0.404000
Ellapsed seconds: 0.505000
Ellapsed seconds: 0.606000
Ellapsed seconds: 0.707000
Ellapsed seconds: 0.808000
Ellapsed seconds: 0.909000
Ellapsed seconds: 1.010000
```

System::eval()

Evals an expression.

Syntax

```
System::eval(_expression_,_args)
```

Parameters

- *_expresion*: String as the expression to be evaluated.
- *_args* (optional): Arguments that will be used as variables within the evaluation.

Returns

Example

```
Console::outln(System::eval(
    "return op1+op2;\n"
    ,{
        op1:5
        ,op2:10
    })
);
```

Console output:

```
15
```

System::assert()

Evals an assert and raise an error if an boolean expression is *false*.

Syntax

```
System::assert(_expression,_message)
```

Parameters

- *_expression*: *Boolean* expression to be evaluated.
- __message (options): the message about the error. If message is not provided, a default message is assigned.

Returns

Example

```
var m=30
System::assert(m < 20,"n > m")
```

Console output:

```
Assert error :n > m
```

System::error

Raises custom error by user.

Syntax

```
System::error(_message,..._args)
```

Parameters

- *_message*: the message to print as the error. The message representation can be specify arguments and/or its format. Argument format is defined in the string within curly brackets (i.e {}) where it specifies argument number and its format (see String::format()).
- *_args* (optional): variable args to be replaced in the string representation.

Returns

Example

```
var m=30
if(!(m < 20)){
    System::error("Error !(m < 20). 'm' is {0}", m)
}
```

Console output:

```
Error !(m < 20). 'm' is 30
```

2.11.3. Math

Math module implements a set of functions and properties related with maths.

Static member properties

Math::PI

The number PI.

Syntax

```
Math::PI
```

Example

```
Console::outln(Math::PI)
```

Console output:

```
3.141593
```

Static functions

Math::sin()

Performs a sinus.

Syntax

```
Math::sin(_radians)
```

Parameters

- *_radians*: Radians value.

Returns

A *Float* value as result of sinus of *_radians* between 0..1.

Example

```
Console::outln("The sine of 30 degrees is "+Math::sin(Math::degToRad(30)))
```

Console output:

```
The sine of 30 degrees is 0.500000
```

Math::cos()

Performs a cosinus.

Syntax

```
Math::cos(_radians)
```

Parameters

- *_radians*: Radians value.

Returns

A *Float* value as result of cosinus of *_radians* between 0..1.

Example

```
Console::outln("The cosine of 60 degrees is "+Math::cos(Math::degToRad(60)))
```

Console output:

```
The cosine of 60 degrees is 0.500000
```

Math::abs()

Performs a absolute.

Syntax

```
Math::abs(_value)
```

Parameters

- _value: Numeric input value

Returns

A *Float* value as the absolute value.

Example

```
Console::outln("The absolute value of 3.1416 is " + Math::abs(3.1416) );
Console::outln("The absolute value of -10.6 is " + Math::abs(-10.6) );
```

Console output:

```
The absolute value of 3.1416 is 3.141600
The absolute value of -10.6 is 10.600000
```

Math::pow()

Performs a power operation.

Syntax

```
Math::pow(_base,_exponent)
```

Parameters

- _base: Base value
- _exponent: Exponent value

Returns

A *Float* value as the result of raising *_base* to the *_power* exponent.

Example

```
Console::outln("7 ^ 3 = " + Math::pow (7.0, 3.0) );
Console::outln("4.73 ^ 12 = "+ Math::pow (4.73, 12.0) );
Console::outln("32.01 ^ 1.54 = "+ Math::pow (32.01, 1.54) );
```

Console output:

```
7 ^ 3 = 343.000000
4.73 ^ 12 = 125410448.000000
32.01 ^ 1.54 = 208.036652
```

Math::degToRad()

Converts degres to radiants.

Syntax

```
Math::degToRad(_degrees)
```

Parameters

- *_degrees*: Degrees value

Returns

A *Float* value as the result of *_degrees* as radians

Example

```
Console::outln("Math::degToRad(30) => "+Math::degToRad(30))
```

Console output:

```
Math::degToRad(30) => 0.523599
```

Math::random()

Returns random float number between [0..1).

Syntax

```
Math::random()
```

Parameters

None

Returns

A *Float* value between [0..1)

Example

```
for(var i=0; i < 5; i++){
    Console::outln("random value : "+Math::random());
}
```

Console output:

```
random value : 0.498276
random value : 0.584765
random value : 0.520585
random value : 0.372692
random value : 0.600879
```

Math::max()

Returns maximmum of two input values.

Syntax

```
Math::max(_v1,_v2)
```

Parameters

- *_v1*: First value
- *_v2*: Second value

Returns

The maximum of *v1_ and* v2_

Example

```
Console::outln("The maximum of 20 and 30 is : "+Math::max(20,30))
```

Console output:

```
The maximum of 20 and 30 is : 30.000000
```

Math::min()

Returns minimmum of two input values.

Syntax

```
Math::min(_v1,_v2)
```

Parameters

- *_v1*: First value
- *_v2*: Second value

Returns

The minimmum of *v1_ and* v2_

Example

```
Console::outln("The minimum of 20 and 30 is : "+Math::min(20,30))
```

Console output:

```
The minimum of 20 and 30 is : 20.000000
```

Math::sqrt()

Computes the square root of input value.

Syntax

```
Math::sqrt(_value)
```

Parameters

- *_value*: Input value

Returns

A *Float* value as the result of the square root of *_value*

Example

```
Console::outln ("Math::sqrt(1024.0) = "+ Math::sqrt(1024.0));
```

Console output:

```
Math::sqrt(1024.0) = 32.000000
```

Math::floor()

Rounds a value downward.

Syntax

```
Math::floor(_value)
```

Parameters

- *_value*: Value to round down.

Returns

A *Float* value as *_x* rounded downward

Example

```
 Console::outln( "floor of 2.3 is "+Math::floor(2.3) );
 Console::outln( "floor of 3.8 is "+Math::floor(3.8) );
 Console::outln( "floor of -2.3 is "+Math::floor(-2.3) );
 Console::outln( "floor of -3.8 is "+Math::floor(-3.8) );
```

Console output:

```
floor of 2.3 is 2.000000
floor of 3.8 is 3.000000
floor of -2.3 is -3.000000
floor of -3.8 is -4.000000
```

Math::ceil()

Rounds a value upward.

Syntax

```
Math::ceil(_x)
```

Parameters

- *_x*: Value to round up.

Returns

A *Float* as the largest integral value that is not greater than *_x*

Example

```
Console::outln( "ceil of 2.3 is "+Math::ceil(2.3) );
Console::outln( "ceil of 3.8 is "+Math::ceil(3.8) );
Console::outln( "ceil of -2.3 is "+Math::ceil(-2.3) );
Console::outln( "ceil of -3.8 is "+Math::ceil(-3.8) );
```

Console output:

```
ceil of 2.3 is 3.000000
ceil of 3.8 is 4.000000
ceil of -2.3 is -2.000000
ceil of -3.8 is -3.000000
```

Math::round()

Rounds a value to the nearest integral.

Syntax

```
Math::round(_value)
```

Parameters

- *_x*: Value to round.

Returns

A *Float* value as *_x* to the nearest integral.

Example

```
Console::outln("round of 2.3 is " + Math::round(2.3) );
Console::outln("round of 3.8 is " + Math::round(3.8) );
Console::outln("round of 5.5 is " + Math::round(5.5) );
Console::outln("round of -2.3 is "+ Math::round(-2.3) );
Console::outln("round of -3.8 is "+ Math::round(-3.8) );
Console::outln("round of -5.5 is "+ Math::round(-5.5) );
```

Console output:

```
round of 2.3 is 2.000000
round of 3.8 is 4.000000
round of 5.5 is 6.000000
round of -2.3 is -2.000000
round of -3.8 is -4.000000
round of -5.5 is -6.000000
```

2.11.4. Json

Json module implements a set of functions about json format.

Json::serialize()

Serializes value or object to json string.

Syntax

```
Json::serialize(_object, _format)
```

Parameters

- *_object*: *Object* to serialize.
- *_format* (optional): It formats serialized string.

Returns

A *String* as the result of the serialization.

Example

```
var object={
    encoding : "UTF-8"
    ,number: 3.34E-5
    ,plug_ins : [
        "python",
        "c++",
        "ruby"
      ]
    ,indent : { "length" : 3, "use_space": true }
}

Console::outln("serialize :")
Console::outln()
Console::outln(Json::serialize(object))
Console::outln()
Console::outln("serialize with format : ")
Console::outln()
Console::outln(Json::serialize(object,true))
```

Console output:

```
serialize :

{"encoding":"UTF-8","number":0.000033,"plug_ins":["python","c++","ruby"],"indent":{"length":3,"use_space":true}}

serialize with format :

{
    "encoding":"UTF-8"
    ,"number":0.000033
    ,"plug_ins":["python","c++","ruby"]
    ,"indent":{
        "length":3
        ,"use_space":true
    }
}
```

Json::deserialize()

Deserializes json string.

Syntax

```
Json::deserialize(_json_string)
```

Parameters

- *_json_string*: Json string value.

Returns

A *Object* as a result of the deserialization.

Example

```
var json_string="{\n"+
        "\"encoding\" : \"UTF-8\",\n"+
        "\"number\": 3.34E-5\n"+
        ",\"plug_ins\" : [\n"+
            "\"python\",\n"+
            "\"c++\",\n"+
            "\"ruby\"\n"+
          "]\n"+
        ",\"indent\" : { \"length\" : 3, \"use_space\": true }\n"+
    "}"

var deseriazed_json_string=Json::deserialize(json_string);
Console::outln("Deserialized json result:\n")
for(var k,v in deseriazed_json_string){
    Console::outln("key => '{0}' value => {1} ",k,v)
}
```

Console output:

```
Deserialized json result:

key => 'encoding' value => UTF-8
key => 'number' value => 0.000033
key => 'plug_ins' value => ["python","c++","ruby"]
key => 'indent' value => {"length":3.000000,"use_space":true}
```

2.11.5. TimeSpan

TimeSpan represents a time interval that is difference between two times measured in number of *days*, *hours*, *minutes*, and *seconds*. *TimeSpan* is used to compare two *DateTime* (see in section DateTime) objects to find the difference between two dates.

Member properties

Datetime has the following member properties,

Variable	description
`days`	days
`hours`	hours
`minutes`	minutes
`seconds`	seconds

2.11.6. DateTime

DateTime represents an instant in time as a date and time of day.

Static functions

DateTime::_sub()

Perform sub (aka -) operation between two objects type *DateTime*.

Syntax

```
_sub(_op1,_op2)
```

Parameters

- *_op1*: First operand type DateTime as the minuend.
- *_op2*: Second operand type DateTime as subtrahend.

Returns

A *TimeSpan* a result of *_op1* - *_op2*.

Example

```
var birth=new DateTime(1979,9,6,12,00,00)
Console::outln("birth => "+birth)
var now=DateTime::now()
var diff=now-birth;
Console::outln("years => "+Integer::parse(diff.days/365))
```

Console output:

```
birth => 1979-09-06 12:00:00
years => 44
```

DateTime::now()

Creates a new *DateTime* object at LOCAL time.

Syntax

```
DateTime::now()
```

Example

```
Console::outln("DateTime::now() => " + DateTime::now())
```

Console output:

```
DateTime::now() => 2024-05-07 12:32:11
```

DateTime::nowUtc()

Creates a new *DateTime* object at UTC time.

Syntax

```
DateTime::nowUtc()
```

Example

```
Console::outln("DateTime::now() => " + DateTime::now())
Console::outln("DateTime::nowUtc() => "+DateTime::nowUtc())
```

Console output:

```
DateTime::now() => 2024-05-07 12:32:11
DateTime::nowUtc() => 2024-05-07 10:32:11
```

Member function

DateTime::constructor()

DateTime constructor.

Syntax

```
constructor(_year,_month,_day,_hour,_minute,_second)
```

Parameters

- *_year*: Integer of the year
- *_month*: Integer in [1..12] as month
- *_day*: Integer in [1..31] as day
- *_hour*: Integer in [0..23] as hour
- *_minute*: Integer in [0..59] as minute
- *_second*: Integer in [0..59] as second

Returns

A *DateTime* object.

Example

```
Console::outln(new DateTime(1979,9,6,12,00,00))
```

Console output:

```
1979-09-06 12:00:00
```

Member properties

Datetime has the following member properties,

Variable	description
week_day	The week's day
month_day	The month's day
year_day	The year's day
second	seconds
minute	minutes
hour	hours
day	day
month	month
year	year

Example

```
var now = DateTime::now();
Console::outln("now => "+ now);
Console::outln("now.week_day => "+ now.week_day);
Console::outln("now.month_day => "+ now.month_day);
Console::outln("now.year_day => "+ now.year_day);
Console::outln("now.second => "+ now.second);
Console::outln("now.minute => "+ now.minute);
Console::outln("now.hour => "+ now.hour);
Console::outln("now.day => "+ now.day);
Console::outln("now.month => "+ now.month);
Console::outln("now.year => "+ now.year);
```

Console output:

```
now => 2024-05-07 12:32:11
now.week_day => 2
now.month_day => 7
now.year_day => 127
now.second => 11
now.minute => 32
now.hour => 12
now.day => 7
now.month => 5
now.year => 2024
```

Member functions

DateTime::addSeconds()

Adds seconds to datetime object.

Syntax

```
DateTime::addSeconds(_seconds)
```

Parameters

- *_seconds*: *Integer* value or variable to add as seconds.

Returns

None

Example

```
var now = DateTime::now();
Console::outln("now => {0}", now);
now.addSeconds(10);
Console::outln("after +10 seconds from now => {0}", now);
```

Console output:

```
now => 2024-05-07 12:32:10
after +10 seconds from now => 2024-05-07 12:32:20
```

DateTime::addMinutes()

Adds minutes to datetime object.

Syntax

```
DateTime::addMinutes(_minutes)
```

Parameters

- *_minutes*: *Integer* value or variable to add as minutes.

Returns

None

Example

```
var now = DateTime::now();
Console::outln("now => {0}", now);
now.addMinutes(10);
Console::outln("after +10 minutes from now => {0}", now);
```

Console output:

```
now => 2024-05-07 12:32:10
after +10 minutes from now => 2024-05-07 12:42:10
```

DateTime::addHours()

Adds hours to datetime object.

Syntax

```
DateTime::addHours(_hours)
```

Parameters

- *_hours*: *Integer* value or variable to add as hours.

Returns

None

Example

```
var now = DateTime::now();
Console::outln("now => {0}", now);
now.addHours(4);
Console::outln("after +4 hours from now => {0}", now);
```

Console output:

```
now => 2024-05-07 12:32:10
after +4 hours from now => 2024-05-07 16:32:10
```

DateTime::addDays()

Adds days to datetime object.

Syntax

```
DateTime::addDays(_days)
```

Parameters

- *_days*: *Integer* value or variable to add as days.

Returns

None

Example

```
var now = DateTime::now();
Console::outln("now => {0}", now);
now.addDays(36);
Console::outln("after +36 days from now => {0}", now);
```

Console output:

```
now => 2024-05-07 12:32:10
after +36 days from now => 2024-06-12 12:32:10
```

DateTime::addMonths()

Adds months to datetime object.

Syntax

```
DateTime::addMonths(_months)
```

Parameters

- *_months*: *Integer* value or variable to add as months.

Returns

None

Example

```
var now = DateTime::now();
Console::outln("now => {0}", now);
now.addMonths(10);
Console::outln("after +10 months from now => {0}", now);
```

Console output:

```
now => 2024-05-07 12:32:10
after +10 months from now => 2025-03-07 12:32:10
```

DateTime::addYears()

Adds years to datetime object.

Syntax

```
DateTime::addYears(_years)
```

Example

```
var now = DateTime::now();
Console::outln("now => {0}", now);
now.addYears(3);
Console::outln("after +3 years from now => {0}", now);
```

Console output:

```
now => 2024-05-07 12:32:11
after +3 years from now => 2027-05-07 12:32:11
```

Chapter 3. The API

This section aims to explain ZetScript API. It will start by explaining the basic data used in the API. Next sections will be like a tutorial how to call C++ from ZetScript, how to call ZetScript from C++ and finally how to expose native types in ZetScript.

3.1. Data types

This section will describe the data types used in the ZetScript API. Some of them are used as helpers and others interops with script environment. Some of types described it has methods and fields that are used in ZetScript internally but this documentation only will explain the relevant ones needed for the user.

3.1.1. zetscript::zs_int

Integer data type it defines a integer variable with range from -(2^{b-1}) to 2^{b-1}-1 where b=32 or b=64 it depending whether ZetScript is compiled for 32bits or 64bits.

Example

```
zetscript::zs_int i=10;
```

3.1.2. zetscript::zs_float

zetscript::zs_float type it defines a float variable represented as IEEE-754 floating point numbers in 32-bit or 64 bit it depending whether ZetScript is compiled for 32bits or 64bits.

Example

```
zetscript::zs_float f=20.5;
```

3.1.3. zetscript::String

String it defines a string represented a sequence of chars

Constructor

The String constructor creates a new string as empty or initialized with other string or String.

Syntax

```
String();
String(const char * _s);
String(const String & _str);
String(String && _str);
```

Parameters

- *_s* : A pointer to an array of characters.
- *_str* : A String object.

Example

```
#include "zetscript.h"

int main(void){

    // empty string
    zetscript::String empty_string;

    // initialized string
    zetscript::String string=zetscript::String("Hello world");

    return 0;
}
```

Static constants

String::n_pos

npos is a static member constant value with the greatest possible value.

Static functions

String::format()

Returns a formatted string.

Syntax

```
String format(const char *_format,...);
```

Parameters

- *_format* : C string that contains the text to be returned as String. It can optionally contain embedded format specifiers that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format specifier follows the same prototype as C printf:

  ```
  %[flags][width][.precision][length]specifier
  ```

Specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	%

The format specifier can also contain sub-specifiers: flags, width, .precision and modifiers (in that order), which are optional and follow these specifications:

flags	description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign. (space) If no sign is going to be written, a blank space is inserted before the value.

#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see width sub-specifier).

width	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	description
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed after the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

The length sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without length specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

Length	Description
hh	For integer types, causes printf to expect an int-sized integer argument which was promoted from a char.
h	For integer types, causes printf to expect an int-sized integer argument which was promoted from a short.
l	For integer types, causes printf to expect a long-sized integer argument. For floating-point types, this is ignored. float arguments are always promoted to double when used in a varargs call.[4]
ll	For integer types, causes printf to expect a long long-sized integer argument.
L	For floating-point types, causes printf to expect a long double argument.
z	For integer types, causes printf to expect a size_t-sized integer argument.
j	For integer types, causes printf to expect a intmax_t-sized integer argument.
t	For integer types, causes printf to expect a ptrdiff_t-sized integer argument.

Return

A formated String

Example

```
#include "zetscript.h"

int main(){

    zetscript::String s;

    printf("%s\n",zetscript::String::format("Characters: %c %c", 'a', 65).toConstChar());
    printf("%s\n",zetscript::String::format("Decimals: %d %ld", 1977, 650000L).toConstChar());
    printf("%s\n",zetscript::String::format("Preceding with blanks: %10d", 1977).toConstChar());
    printf("%s\n",zetscript::String::format("Preceding with zeros: %010d", 1977).toConstChar());
    printf("%s\n",zetscript::String::format("Some different radices: %d %x %o %#x %#o", 100, 100, 100, 100, 100).toConstChar());
    printf("%s\n",zetscript::String::format("floats: %4.2f %+.0e %E", 3.1416, 3.1416, 3.1416).toConstChar());
    printf("%s\n",zetscript::String::format("Width trick: %*d", 5, 10).toConstChar());
    printf("%s\n",zetscript::String::format("%s", "A string").toConstChar());
    printf("%s\n",zetscript::String::format("%10s", "A limited string").toConstChar());

    return 0;
}
```

Console output:

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:       1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+00 3.141600E+00
Width trick:    10
A string
A limited string
```

Member functions

String::append()

Appends a character o string at the end of current contents.

Syntax

```
void append(char _c);
void append(const char * _s);
void append(const char *_s, int _len);
void append(const String & _str);
```

Parameters

- *_c* : The character to append.
- *_s* : A pointer to array of characters.
- *_len* : The length of number of characters to copy.
- *_str* : A String object.

Return

None

Example

```
#include "zetscript.h"
int main(){
    zetscript::String string;

    string="Hello";
    string.append(" World");
    string.append('!');

    printf("%s\n",string.toConstChar());

    return 0;
}
```

Console output:

```
Hello World!
```

String::at()

Returns a reference to the character at position *_pos*.

Syntax

```
char& at (int _pos);
const char& at (int _pos) const;
```

Parameters

- *_pos* : The integer value as the position.

Return

The character at the specified position in the string.

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="Hello World";

    for(int c=0; c < string.length();c++){
        printf("%c",string.at(c));
    }

    printf("\n");

    return 0;
}
```

Console output:

```
Hello World
```

String::clear()

Erases the contents of the string.

Syntax

```
void clear();
```

Parameters

None

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="Hello World";

    string.clear();

    printf("string : '%s'\n",string.toConstChar());

    return 0;
}
```

Console output:

```
string : ''
```

String::contains()

Finds out whether a string contains a substring or not.

Syntax

```
bool String::contains(const String & _str);
```

Parameters

- *_str*: A substring to find.

Returns

true if the string *_str* exist, false otherwise.

Example

```
#include "zetscript.h"

int main(){

    zetscript::String s="The quick brown fox jumps over the lazy dog.";

    printf("s.contains(\"fo\") => '%s'\n",s.contains("fo")?"true":"false");
    printf("s.contains(\"foy\") => '%s'\n",s.contains("foy")?"true":"false");
    return 0;
}
```

Console output:

```
s.contains("fo") => 'true'
s.contains("foy") => 'false'
```

String::endsWith()

Returns whether the string ends with the specified character(s).

Syntax

```
bool startsWith() const;
```

Parameters

None

Return

true if the string ends with the specified character(s).

Example

```
#include "zetscript.h"

int main(){

    zetscript::String s="Hello World";

    printf("s.endsWith(\"Hel\")  => '%s \n"
    ,s.endsWith("Hel")?"true":"false");

    printf("s.endsWith(\"orld\")  => '%s' \n"
    ,s.endsWith("orld")?"true":"false");


    return 0;
}
```

Console output:

```
s.endsWith("Hel")  => 'false
s.endsWith("orld")  => 'true'
```

String::erase()

Erases part of the string, reducing its length.

Syntax

```
void erase(int _pos, int _len);
void erase(int _pos);
```

Parameters

- *_#pos* : Position of the first character to be erased.
- *_len* : Number of characters to erase.

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="Helilo World";

    string.erase(3);
    printf("string.erase(3) => '%s'\n",string.toConstChar());

    return 0;
}
```

Console output:

```
string.erase(3) => 'Hello World'
```

String::find()

Searches the string for the first occurrence of the sequence specified by its arguments.

Syntax

```
int find(const String & _str, int _pos = 0) const;
int find(const char *_s, int _pos = 0) const;
```

Parameters

- *_s* : A pointer to an array of characters.
- *_str* : A String object.
- *_pos* : Position of the first character in the string to be considered in the search.

Return

The position of the first character of the first match. If no matches were found, the function returns String::npos

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="the quick brown fox jumps over the lazy dog.";

    printf("string.find(\"the\") => %i\n",string.find("the"));
    printf("string.find(\"fox\") => %i\n",string.find("fox"));

    return 0;
}
```

Console output:

```
string.find("the") => 0
string.find("fox") => 16
```

String::findLastOf()

Searches the string for the last character that matches any of the characters specified in its arguments.

Syntax

```
int findLastOf(const char *_s, int _pos = npos) const;
int findLastOf(const String & _str, int _pos = npos) const;
```

Parameters

- *_str* : A String object.
- *_s* : A pointer to an array of characters.
- *_pos* : Position of the last character in the string to be considered in the search.

Return

The position of the last character that matches. If no matches are found, the function returns string::npos.

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="the fox jumps over the lazy dog.";

    printf("string.findLastOf(\"the\") => %i\n",string.findLastOf("fo"));

    return 0;
}
```

Console output:

```
string.findLastOf("the") => 29
```

String::getSubstring()

Returns a new string with its value initialized to a copy of a substring of this object.

Syntax

```
String  getSubstring (int _pos = 0, int _len = npos) ;
```

Parameters

- *_pos* : Position of the first character to be copied as a substring.
- *_len* : Number of characters to include in the substring. As default it passes String::npos that indicates it will take all characters until the end of the string.

Return

A string with a substring of this object.

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="Hello World";

    printf("string.substring(0) => '%s'\n",string.getSubstring(0).toConstChar());
    printf("string.substring(3) => '%s'\n",string.getSubstring(3).toConstChar());
    printf("string.substring(2,3) => '%s'\n",string.getSubstring(2,3).toConstChar());

    return 0;
}
```

Console output:

```
string.substring(0) => 'Hello World'
string.substring(3) => 'lo World'
string.substring(2,3) => 'llo'
```

String::insert()

Inserts additional characters into the string right before the character indicated by *_pos*.

Syntax

```
void insert(int _pos, char _char);
void insert(int _pos, const String & _string);
```

Parameters

- *_pos* : Position of the first character to insert.
- *_c* : The character to insert.
- *_str* : The characters from String object to insert.

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="Hello Wd";

    string.insert(7,"orl");
    printf("string.insert(7,\"orl\") => '%s'\n",string.toConstChar());

    return 0;
}
```

Console output:

```
string.insert(7,"orl") => 'Hello World'
```

String::isEmpty()

Returns whether the string is empty.

Syntax

```
bool empty() const;
```

Parameters

None

Return

true if the string length is 0, false otherwise.

Example

```
#include "zetscript.h"

int main(){

    zetscript::String empty_string;
    zetscript::String non_empty_string="Hello World";

    printf("empty_string.empty()  => '%s' \n"
    ,empty_string.isEmpty()?"true":"false");

    printf("non_empty_string.empty() => '%s' \n"
    ,non_empty_string.isEmpty()?"true":"false");

    return 0;
}
```

Console output:

```
empty_string.empty()  => 'true'
non_empty_string.empty() => 'false'
```

String::length()

Returns the length of the string, in terms of bytes.

Syntax

```
int length() const;
```

Parameters

None

Return

The number of bytes in the string.

Example

```
#include "zetscript.h"
int main(){
    zetscript::String string="Hello World";

    printf("the length of '%s' is %i\n",string.toConstChar(),string.length());

    return 0;
}
```

Console output:

```
the length of 'Hello World' is 11
```

String::operator=()

Assigns a new value to the string, replacing its current contents.

Syntax

```
String& operator=(const String & _str);
String& operator=(String && _str);
String& operator=(const char* _s);
```

Parameters

- *_s* : A pointer to an array of characters.
- *_str* : A String object.

Return

*this

Example

```
#include "zetscript.h"
int main(){
    zetscript::String string;

    string="Hello world";

    printf("%s\n",string.toConstChar());

    return 0;
}
```

Console output:

```
Hello world
```

String::operator+=()

Appends string at the end of current contents.

Syntax

```
String& operator+=(const String& _string);
String& operator+=(const char* _string);
String& operator+=(char _char);
```

Parameters

- *_s* : A pointer to an array of characters.
- *_str* : String object.

Return

*this

Example

```
#include "zetscript.h"
int main(){
    zetscript::String string;

    string="Hello";
    string+=" World";
    string+='!';

    printf("%s",string.toConstChar());

    return 0;
}
```

Console output:

```
Hello World!
```

String::operator[]()

Returns a reference to the character from a position.

Syntax

```
char& operator[] (int _pos);
const char& operator[] (int _pos) const;
```

Parameters

- *_pos* : The integer value as the position.

Return

The character at the specified position in the string.

Example

```
#include "zetscript.h"
int main(){
    zetscript::String string="Hello World";

    for(int c=0; c < string.length();c++){
        printf("%c",string[c]);
    }

    printf("\n");

    return 0;
}
```

Console output:

```
Hello World
```

String::split()

Splits the string into multiple strings in an array by a char or string delimiter.

Syntax

```
String split(char _delimiter) const;
String split(const String & _delimiter) const;
```

Parameters

- delimiter : A character or string as the delimiter

Return

A vector representing the strings split by the delimiter.

Example

```
#include "zetscript.h"

int main(){
    zetscript::String s="The quick brown fox jumps over the lazy dog.";

    zetscript::Vector<zetscript::String> v=s.split(' ');

    for(int i=0; i < v.length();i++ ){
        printf("%s\n",v.get(i).toConstChar());
    }
}
```

Console output:

```
The
quick
brown
fox
jumps
over
the
lazy
dog.
```

String::startsWith()

Returns whether the string starts with the specified character(s).

Syntax

```
bool startsWith() const;
```

Parameters

None

Return

true if the string starts with the specified character(s).

Example

```
#include "zetscript.h"

int main(){

    zetscript::String s="Hello World";

    printf("s.startsWith(\"Hel\")  => '%s \n"
    ,s.startsWith("Hel")?"true":"false");

    printf("s.startsWith(\"orld\")  => '%s' \n"
    ,s.startsWith("orld")?"true":"false");

    return 0;
}
```

Console output:

```
s.startsWith("Hel")  => 'true
s.startsWith("orld")  => 'false'
```

String::toConstChar()

Returns a pointer to an array that contains a null-terminated sequence of characters (i.e., a C-string) representing the current value of the string object.

Syntax

```
const char * toConstChar() const;
```

Parameters

None

Return

A pointer of characters of the string object's value.

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="Hello World";

    printf("string.toConstChar() => '%s'\n",string.toConstChar());

    return 0;
}
```

Console output:

```
string.toConstChar() => 'Hello World'
```

String::toLowerCase()

Returns current string as lowe case

Syntax

```
String toLowerCase() const;
```

Parameters

None

Return

A String value, representing the new string converted to lower case

Example

```
#include "zetscript.h"

int main(){
    zetscript::String s="Hello World";

    printf("s.toLowerCase() => '%s'",s.toLowerCase().toConstChar());
    return 0;
}
```

Console output:

```
s.toLowerCase() => 'hello world'
```

String::toUpperCase()

Returns current string as upper case

Syntax

```
String toUpperCase() const;
```

Parameters

None

Return

A String value, representing the new string converted to upper case.

Example

```
#include "zetscript.h"

int main(){
    zetscript::String s="Hello World";

    printf("s.toUpperCase() => '%s'",s.toUpperCase().toConstChar());
    return 0;
}
```

Console output:

```
s.toUpperCase() => 'HELLO WORLD'
```

String::setSubstring()

Replaces a portion of the string.

Syntax

```
String & setSubstring(int _pos, int _len, const String & _to_replace);
```

Parameters

- *_pos* : Position of the first character to be replaced.
- *_len* : Number of characters to replace. A value of string::npos indicates all characters until the end of the string.
- *_str* : A String object.

Return

*this

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="this is a test string.";
    zetscript::String str2="n example";

    string.setSubstring(9,5,str2);

    printf("string.replace(9,5,str2) => '%s'\n",string.toConstChar());

    return 0;
}
```

Console output:

```
string.replace(9,5,str2) => 'this is an example string.'
```

Static functions

String::operator+()

Returns a new string object with its value being the concatenation of the characters in left operand followed by those of right operand.

Syntax

```
friend String operator+(const String & _s1, const String &_s2);
friend String operator+(const String & _s1, const char *_s2);
friend String operator+(const char * _s1, const String & _s2);

friend String operator+(const String & _s1, char _s2);
friend String operator+(char  _s1, const String & _s2);
```

Parameters

- *_s1* : A char, pointer to array of characters or String object as left operand.
- *_s2* : A char, pointer to array of characters or String object as right operand.

Return

A string whose value is the concatenation of _s1 and _s2.

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string1="Hello",string2="World";
    zetscript::String string=string1 + " "+ string2 + "!!";

    printf("%s\n",string.toConstChar());

    return 0;
}
```

Console output:

```
Hello World!!
```

Relational operators

String implements the EQUAL (aka ==) and NOT EQUAL (aka !=) comparison operation between the left operand and right operand.

Syntax

```
friend bool operator _REL_OPERATOR_(const String & _s1, const String &_s2);
friend bool operator _REL_OPERATOR_(const String & _s1, const char *_s2);
friend bool operator _REL_OPERATOR_(const char * _s1, const String & _s2);
```

Where _REL_OPERATOR_ can be == or **!=**

Parameters

- *_s1* : A pointer to array of characters or String object as left operand.
- *_s2* : A pointer to array of characters or String object as right operand.

Return

true if satisfies the condition _s1 _R _s2 and false otherwise.

Example

```
#include "zetscript.h"

int main(){
    zetscript::String string="Hello World";
    zetscript::String string2="!!Hello World!!";

    if(string == "Hello World"){
        printf("string == 'Hello World'\n");
    }

    if(string2 != "Hello World"){
        printf("string2 != 'Hello World'\n");
    }

    return 0;
}
```

Console output:

```
string == 'Hello World'
string2 != 'Hello World'
```

3.1.4. zetscript::Vector

Vector it defines a unidimensional vector of elements of type defined on its template parameter.

Constructor

The Vector constructor creates a new Vector as empty or initialized with other Vector.

```
Vector();
Vector(const Vector & _vector);
```

Parameters

- *_vector* : A vector object.

Example

```
#include "zetscript.h"

int main(){

    // empty vector of ints
    zetscript::Vector<int> vector;

    return 0;
}
```

Static constants

Vector::n_pos

npos is a static member constant value with the greatest possible value.

Member functions

Vector::clear()

Erases the contents of the vector.

Syntax

```
void        clear();
```

Parameters

None

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    for(int i=0; i < 10; i++){
        vector.push(i);
    }

    vector.clear();

    printf("vector.length() => %i\n",vector.length());

    return 0;
}
```

Console output:

```
vector.length() => 0
```

Vector::concat()

Copies all elements from other vector at the end of current contents.

Syntax

```
void        concat(const Vector<_T>  & _vector);
```

Parameters

- *_vector* : A vector object.

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector1;
    zetscript::Vector<int> vector2;
    zetscript::Vector<int> vector3;

    for(int i=0; i < 2; i++){
        vector1.push(i);
    }

    for(int i=2; i < 4; i++){
        vector2.push(i);
    }
    vector3.concat(vector1);
    vector3.concat(vector2);

    printf("vector3 contents : [");

     for(int i=0; i < vector3.length(); i++){
        if(i>0){
            printf(",");
        }
        printf("%i",vector3.get(i));
    }

    printf("]\n");

    return 0;
}
```

Console output:

```
vector3 contents : [0,1,2,3]
```

Vector::data()

Returns the pointer to the array of the elements.

Syntax

```
_T *data();
```

Parameters

None

Return

A pointer of elements of the vector.

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    for(int i=0; i < 5; i++){
        vector.push(i);
    }

    int *ptr_data=vector.data();

    printf("contents from ptr_data : [");

    for(int i=0; i < vector.length(); i++){
        if(i>0){
            printf(",");
        }
        printf("%i",*ptr_data++);
    }

    printf("]\n");

    return 0;
}
```

Console output:

```
contents from ptr_data : [0,1,2,3,4]
```

Vector::erase()

Erases an element from a position.

Syntax

```
void       erase( int  _pos);
```

Parameters

- *_pos* : Position of the element to be erased.

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    for(int i=0; i < 5; i++){
        vector.push(i);
    }

    vector.erase(2);

    printf("vector contents  : [");
    for(int i=0; i < vector.length(); i++){
        if(i>0){
            printf(",");
        }
        printf("%i",vector.get(i));
    }

    printf("]\n");

    return 0;
}
```

Console output:

```
vector contents  : [0,1,3,4]
```

Vector::get()

Returns a reference to the element from a position.

Syntax

```
const _T &  get( int  _pos);
```

Parameters

- *_pos* : The integer value as the position.

Return

The element at the specified position in the vector.

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    vector.push(10);
    vector.push(100);
    vector.push(1000);

    printf("vector.get(1) => %i\n",vector.get(1));

    return 0;
}
```

Console output:

```
vector.get(1) => 100
```

Vector::insert()

Inserts an element or copies all elements of other vector from a position.

Syntax

```
void        insert(int _pos,const _T & _element);
void        insert(int _pos,const Vector<_T>  & _vector, int _len=npos);
```

Parameters

- *_pos* : Position of the first element to insert.
- *_vector* : The elements of vector source to insert.
- *_len* : The number of elements to insert. A value of Vector::npos indicates all elements until the end of the vector.

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    vector.push(1);
    vector.push(2);
    vector.push(4);

    // insert integer 3 at position 2
    vector.insert(2,3);

    printf("vector contents  : [");

    for(int i=0; i < vector.length(); i++){
        if(i>0){
            printf(",");
        }
        printf("%i",vector.get(i));
    }

    printf("]\n");

    return 0;
}
```

Console output:

```
vector contents  : [1,2,3,4]
```

Vector::length()

Returns the number of elements of the vector.

Syntax

```
int length() const;
```

Parameters

None

Return

The number of elements in the vector.

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    vector.push(1);
    vector.push(2);
    vector.push(3);

    printf("vector.length() => %i",vector.length());

    return 0;
}
```

Console output:

```
vector.length() => 3
```

Vector::operator=()

Replaces current content by the set of elements from other vector.

Syntax

```
Vector& operator=(const Vector& _vector);
```

Parameters

- *_vector* : A vector object.

Return

*this

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector1,vector2;

    vector1.push(1);
    vector1.push(2);
    vector1.push(3);
    vector1.push(4);

    vector2=vector1;

    printf("vector2 contents  : [");

    for(int i=0; i < vector2.length(); i++){
        if(i>0){
            printf(",");
        }
        printf("%i",vector2.get(i));
    }

    printf("]\n");

    return 0;
}
```

Console output:

```
vector2 contents  : [1,2,3,4]
```

Vector::pop()

Returns the last element by copy and erases the last element.

Syntax

```
void          pop();
```

Parameters

None

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    vector.push(1);
    vector.push(2);
    vector.push(3);
    vector.push(4);

    vector.pop();

    printf("vector contents  : [");

    for(int i=0; i < vector.length(); i++){
        if(i>0){
            printf(",");
        }
        printf("%i",vector.get(i));
    }

    printf("]\n");

    return 0;
}
```

Console output:

```
vector contents  : [1,2,3]
```

Vector::push()

Appends an element at the end of current contents.

Syntax

```
bool        push( const _T & _element);
```

Parameters

- *_element* : The element to append.

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    vector.push(1);
    vector.push(2);
    vector.push(3);

    return 0;
}
```

Vector::resize()

Resizes the vector by a length.

Syntax

```
void        resize(int _len);
```

Parameters

- *_len* : The new capacity of the vector.

Return

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    vector.resize(4);

    for(int i=0; i < vector.length(); i++){
        vector.set(i,i);
    }

    printf("vector contents  : [");

     for(int i=0; i < vector.length(); i++){
        if(i>0){
            printf(",");
        }
        printf("%i",vector.get(i));
    }

    printf("]\n");

    return 0;
}
```

Console output:

```
vector contents  : [0,1,2,3]
```

Vector::set()

Replaces the element from a position.

Syntax

```
void       set( int  _pos_, const _T & _element);
```

Parameters

- *_pos* : The integer value as the position.
- *_element* : The element to replace.

Return

none

Example

```
#include "zetscript.h"

int main(){
    zetscript::Vector<int> vector;

    vector.push(1);
    vector.push(100);
    vector.push(3);

    vector.set(1,2); // vector = [1,2,3]

    return 0;
}
```

3.1.5. zetscript::ScriptType

ScriptType is an type managed by ZetScript that defines a builtin or registered script type. ScriptType has the following fields.

- id: The script type id
- name: The script type name
- native_name: Native name of script type
- properties: A integer value composed of the following mask values,

Name	Description
`SCRIPT_TYPE_PROPERTY_NATIVE_OBJECT_REF`	It tells script type is native
`SCRIPT_TYPE_PROPERTY_NON_INSTANTIABLE`	it tells script type is static or not instantiable

3.1.6. zetscript::Symbol

Symbol is a type managed internally by ZetScript that defines a symbol. The *Symbol* type has the following fields:

- scope : scope where symbol was registered
- name: symbol name
- ref_ptr: Pointer reference that holds information about the instance of symbol type.
- n_params; Number of parameters in case the symbol is refers a function.
- properties: A integer value composed of the following mask values,

Name	Description
SYMBOL_PROPERTY_NATIVE_OBJECT_REF	Tells that the *Symbol* it holds a native pointer
SYMBOL_PROPERTY_STATIC	Tells that the *Symbol* static
SYMBOL_PROPERTY_SCRIPT_TYPE	Tells that the *Symbol* is a *ScriptType*
SYMBOL_PROPERTY_SCRIPT_FUNCTION	Tells that the *Symbol* is a *ScriptFunction*
SYMBOL_PROPERTY_CONST	Tells that the *Symbol* is constant
SYMBOL_PROPERTY_MEMBER_PROPERTY	Tells that the *Symbol* is a *MemberProperty*
SYMBOL_PROPERTY_ARG_BY_REF	Tells that the *Symbol* refers an argument by reference
SYMBOL_PROPERTY_ALLOCATED_STK	Tells that the *Symbol* an allocated StackElement

3.1.7. zetscript::StackElement

StackElement is the data type in the stack used by the VirtualMachine. StackElement has the following fields:

value: The value of StackElement _properties : Defines the content of value by the following mask values,

Value	Description
`STACK_ELEMENT_PROPERTY_UNDEFINED`	It tells that *StackElement* is **Undefined**
`STACK_ELEMENT_PROPERTY_NULL`	It tells that *StackElement* is **Null**
`STACK_ELEMENT_PROPERTY_CHAR_PTR`	It tells that the contents of *StackElement* is a **const char ***
`STACK_ELEMENT_PROPERTY_INT`	It tells that the *StackElement* is a *Boolean* representing its value as **zs_int**
`STACK_ELEMENT_PROPERTY_FLOAT`	It tells that the *StackElement* is a *Float* representing its value as **zs_float ***
`STACK_ELEMENT_PROPERTY_BOOL`	It tells that the *StackElement* is a *Boolean* representing its value as *zetscript::zs_int*
`STACK_ELEMENT_PROPERTY_SCRIPT_TYPE_ID`	It tells that the *StackElement* is a script_type_id_ of the *ScriptType* representing its value by a *zetscript::zs_int*
`STACK_ELEMENT_PROPERTY_FUNCTION`	It tells that the *StackElement* is a *ScriptFunction*. Its value is **Symbol *** as the symbol that refers a **Function *** of a function in *ref_ptr* field
`STACK_ELEMENT_PROPERTY_MEMBER_FUNCTION`	It tells that the *StackElement* is a *MemberFunction*. Its value is a **Symbol *** as the symbol and the *MemberFunction* reference is get from *Symbol::ref_ptr* field
`STACK_ELEMENT_PROPERTY_MEMBER_PROPERTY`	It tells that the contents of *StackElement* is a *StackElementMemberProperty* *
`STACK_ELEMENT_PROPERTY_OBJECT`	It tells that the contents of *StackElement* is a **ScriptObject ***
`STACK_ELEMENT_PROPERTY_CONTAINER_SLOT`	It tells that the contents of *StackElement* is a **ContainerSlot ***
`STACK_ELEMENT_PROPERTY_PTR_STK`	It tells that the contents of *StackElement* is a **StackElement ***
`STACK_ELEMENT_PROPERTY_READ_ONLY`	It tells that the *StackElement* is defined as read only

3.1.8. zetscript::ScriptObject

ScriptObject defines a the parent class of any object in TypeScript. This type is usually used internally by ZetScript. The *ScriptObject* type has the following fields:

- script_type_id: The script type representated by the instance of script object. ZetScript has a primitive types which *script_type_ids* are the following,

Name	Description
SCRIPT_TYPE_ID_STRING_SCRIPT_OBJECT	Tells that the object is an instance of StringScriptObject
SCRIPT_TYPE_ID_ARRAY_SCRIPT_OBJECT	Tells that the object is an instance of ObjectArray
SCRIPT_TYPE_ID_OBJECT_SCRIPT_OBJECT	Tells that the object is an instance of ObjectScriptObject

- properties: A integer value composed of the following mask values,

Value	Description
SCRIPT_OBJECT_PROPERTY_CONSTANT	Tells that *ScriptObject* is constant

3.1.9. zetscript::StringScriptObject

StringScriptObject defines a string script object.

Constructor

The StringScriptObject constructor creates a new string object as empty or initialized with a string. StringScriptObject constructor requires a *ScriptEngine* instance as a parameter.

Syntax

```
StringScriptObject(ScriptEngine *_script_engine, const String & _str="");
```

Parameters

- *_script_engine* : ScriptEngine instance.
- *_str* : String value as a initialize value (empty by default).

Example

```
#include "zetscript.h"

zetscript::StringScriptObject *returnNewStringScriptObject(
    zetscript::ScriptEngine *_script_engine
){

    // instance new StringScriptObject using ScriptEngine instance
    zetscript::StringScriptObject *string_object=new zetscript::StringScriptObject(_script_engine,"Hello World!");

    // ...

    return string_object;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers returnNewStringScriptObject
    script_engine.registerFunction("returnNewStringScriptObject",returnNewStringScriptObject);

    // prints the returning of new string object
    script_engine.compileAndRun("Console::outln(returnNewStringScriptObject())");

    return 0;
}
```

Console output:

```
Hello World!
```

Member functions

StringScriptObject::get()

Gets its current string value as String or array of characters

Syntax

```
const String & get();
```

Parameters

None

Return

Its contents as string as String

Example

```
#include "zetscript.h"

void printString(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::StringScriptObject *_string_object
){
    printf("%s\n",_string_object->get().toConstChar());
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers printString
    script_engine.registerFunction("printString",printString);

    // calls printString
    script_engine.compileAndRun("printString(\"Hello World!\")");

    return 0;
}
```

Console output:

```
Hello World!
```

StringScriptObject::getConstChar()

Gets its current string value as array of characters

Syntax

```
const char *getConstChar();
```

Parameters

None

Return

Its contents as string as array of characters

Example

```
#include "zetscript.h"

void printString(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::StringScriptObject *_string_object
){
    printf("%s\n",_string_object->getConstChar());
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers printString
    script_engine.registerFunction("printString",printString);

    // calls printString
    script_engine.compileAndRun("printString(\"Hello World!\")");

    return 0;
}
```

Console output:

```
Hello World!
```

StringScriptObject::length()

Returns the length of the string, in terms of bytes.

Syntax

```
virtual int length();
```

Parameters

None

Return

The number of bytes in the string.

```
#include "zetscript.h"

void printStringLength(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::StringScriptObject *_string_object
){
    printf("The length of '%s' is %i\n",_string_object->toString().toConstChar(),_string_object->length());
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers printStringLength
    script_engine.registerFunction("printStringLength",printStringLength);

    // calls printStringLength
    script_engine.compileAndRun("printStringLength(\"Hello World!\")");

    return 0;
}
```

Console output:

```
The length of 'Hello World!' is 12
```

StringScriptObject::set()

Replaces current string value.

Syntax

```
void set(const String & _str);
```

Parameters

- *_str* : The string value.

Return

None

Example

```
#include "zetscript.h"

zetscript::StringScriptObject *returnString(
    zetscript::ScriptEngine *_script_engine
){
    zetscript::StringScriptObject *string_object = new zetscript::StringScriptObject(_script_engine);
    string_object->set("Hello World!");
    return string_object;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers returnString
    script_engine.registerFunction("returnString",returnString);

    // prints the value returned by returnString
    script_engine.compileAndRun("Console::outln(returnString())");

    return 0;
}
```

Console output:

```
Hello World!
```

StringScriptObject::toString()

Gets its current string value

Syntax

```
String toString();
```

Parameters

None

Return

Its contents as string as String

Example

```
#include "zetscript.h"

void printString(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::StringScriptObject *_string_object
){
    printf("%s\n",_string_object->toString().toConstChar());
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers printString
    script_engine.registerFunction("printString",printString);

    // calls printString
    script_engine.compileAndRun("printString(\"Hello World!\")");

    return 0;
}
```

Console output:

```
Hello World!
```

3.1.10. ArrayScriptObject

ArrayScriptObject it defines a array script object.

Constructor

The ArrayScriptObject constructor creates a new array script object. ArrayScriptObject constructor requires a *ScriptEngine* instance as a parameter.

Syntax

```
ArrayScriptObject(ScriptEngine *_script_engine);
```

Parameters

- *_script_engine* : ScriptEngine instance.

Example

```
#include "zetscript.h"

zetscript::ArrayScriptObject *returnNewArrayScriptObject(
    zetscript::ScriptEngine *_script_engine
){

    // instance new StringScriptObject using ScriptEngine instance
    zetscript::ArrayScriptObject *array_object=new zetscript::ArrayScriptObject(_script_engine);

    //...

    return array_object;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers returnNewArrayScriptObject
    script_engine.registerFunction("returnNewArrayScriptObject",returnNewArrayScriptObject);

    // prints the contents of new array object (empty)
    script_engine.compileAndRun("Console::outln(returnNewArrayScriptObject())");

    return 0;
}
```

Console output:

```
[]
```

Member functions

ArrayScriptObject::elementInstanceOf()

Says whether the element at a position is instance of zetscript data type or a registered type.

Syntax

```
bool    elementInstanceOf<_T>(int _pos);
```

Template

- *_T* : Zetscript data type or a registered type.

Parameters

- *_pos* : The integer value as the position.

Return

Returns true if the element at position *_pos* it type or extends from type *_T*

Example

```
#include "zetscript.h"

// Check whether the array element is type integer, float or string
void testArrayElementInstanceOf(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ArrayScriptObject *_array_object
){

    for(int i=0; i < _array_object->length(); i++){
        printf("Element at position '%i' is type '",i);

        if(_array_object->elementInstanceOf<zetscript::zs_int>(i)){
            printf("Integer");
        }

        if(_array_object->elementInstanceOf<zetscript::zs_float>(i)){
            printf("Float");
        }

        if(_array_object->elementInstanceOf<zetscript::String>(i)){
            printf("String");
        }
        printf("'\n");
    }
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers testArrayElementInstanceOf
    script_engine.registerFunction("testArrayElementInstanceOf",testArrayElementInstanceOf);

    // calls printArrayElementTypes
    script_engine.compileAndRun(
        "testArrayElementInstanceOf(["
            "0"                 // Element at position '1' is type 'Integer'
            ",10.5"             // Element at position '2: is type 'Float'
            ",\"Hello World!\"" // Element at position '3: is type 'String'
        "])");

    return 0;
}
```

Console output:

```
Element at position '0' is type 'Integer'
Element at position '1' is type 'Float'
Element at position '2' is type 'String'
```

ArrayScriptObject::get()

Returns a reference element of type *_T* at position *_pos*.

Syntax

```
_T  get(int _pos);
```

Template

- *_T* : Zetscript data type or a registered type.

Parameters

- *_pos* : The integer value as the position.

Return

The element at the specified position in the array.

Example

```
#include "zetscript.h"

// Print contents of an array.
void printArray(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ArrayScriptObject *_array_object
){
    printf("Array contents : [");
    for(int i=0; i < _array_object->length(); i++){
        if(i>0){
            printf(",");
        }
        if(_array_object->elementInstanceOf<zetscript::zs_int>(i)){
            printf("%i",(int)_array_object->get<zetscript::zs_int>(i));
        }else if(_array_object->elementInstanceOf<zetscript::zs_float>(i)){
            printf("%f",_array_object->get<zetscript::zs_float>(i));
        }else if(_array_object->elementInstanceOf<zetscript::String>(i)){
            printf("'%s'",_array_object->get<zetscript::String>(i).toConstChar());
        }else{
            printf("N/A");
        }
    }
    printf("]\n");
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers 'printArray' function
    script_engine.registerFunction("printArray",printArray);

    // calls printArray
    script_engine.compileAndRun("printArray([0,10.5,\"Hello World\"])");

    return 0;
}
```

Console output:

```
Array contents : [0,10.500000,'Hello World']
```

ArrayScriptObject::length()

Returns the number of the elements of the array.

Syntax

```
int length();
```

Parameters

None

Return

The number of elements in the array.

Example

```
#include "zetscript.h"

void printArrayLength(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ArrayScriptObject *_array_object
){
    printf("The length of vector is %i\n",_array_object->length());
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers printArrayLength
    script_engine.registerFunction("printArrayLength",printArrayLength);

    // calls printArrayLength
    script_engine.compileAndRun("printArrayLength([0,1,2,3])");

    return 0;
}
```

Console output:

```
The length of vector is 4
```

ArrayScriptObject::push()

Appends *_value* of type *_T* at the end of array.

Syntax

```
void push(_T _element);
```

Template

- *_T* : Zetscript data type or a registered type.

Parameters

- *_element* : The element to append.

Return

None

Example

```
#include "zetscript.h"

zetscript::ArrayScriptObject *returnNewArray(
    zetscript::ScriptEngine *_script_engine
){
    // instance new ArrayScriptObject using ScriptEngine instance
    zetscript::ArrayScriptObject *array_object=new zetscript::ArrayScriptObject(_script_engine);

    // push first value as integer 10
    array_object->push<zetscript::zs_int>(10);

    // push second value as float 5.5
    array_object->push<zetscript::zs_float>(5.5);

    // push third value as boolean true
    array_object->push<bool>(true);

    // push 4rth reference string "Hello Word"
    array_object->push<const char *>("Hello World");

    // return object array
    return array_object;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers returnNewArray
    script_engine.registerFunction("returnNewArray",returnNewArray);

    // prints the value returned by returnNewArray
    script_engine.compileAndRun("Console::outln(returnNewArray())");

    return 0;
}
```

Console output:

```
[10,5.500000,true,"Hello World"]
```

ArrayScriptObject::set()

Replaces current value at position *_pos* by *_value* of type *_T*.

Syntax

```
void set<_T>(int _pos, _T _element);
```

Template

- *_T* : Zetscript data type or a registered type.

Parameters

- *_pos* : The integer value as the position.
- *_element* : The element to replace.

Return

None

Example

```
#include "zetscript.h"

void modifyArray(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ArrayScriptObject *_array_object
){

    for(int i=0; i < _array_object->length(); i++){
        switch(i%3){
        case 0: // set a integer
            _array_object->set<zetscript::zs_int>(i,i);
            break;
        case 1: // set a random float
            _array_object->set<zetscript::zs_float>(i,i*10.2);
            break;
        case 2: // set new string
            _array_object->set<zetscript::StringScriptObject *>(i,new zetscript::StringScriptObject(_script_engine,"Hello"));
            break;

        }
    }
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers modifyArray
    script_engine.registerFunction("modifyArray",modifyArray);

    // create a initialized array, it prints. then prints the
    // the array after call 'modifyArray'
    script_engine.compileAndRun(
        "var v=[0,\"hello\",10.0,0,1,2,\"world\"]\n"
        "Console::outln(\"Before call 'modifyArray':{0}\",v);"
        "modifyArray(v)\n"
        "Console::outln(\"After call 'modifyArray':{0}\",v);"
    );

    return 0;
}
```

Console output:

```
Before call 'modifyArray':[0,"hello",10.000000,0,1,2,"world"]
After call 'modifyArray':[0,10.200000,"Hello",3,40.800000,"Hello",6]
```

ArrayScriptObject::toString()

Returns a string as the array contents in json format.

Syntax

```
String toString();
```

Parameters

None

Return

A string of the array contents in json format.

Example

```
#include "zetscript.h"

// print array contents
void printArray(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ArrayScriptObject *_array_object
){
    printf("Array contents : %s",_array_object->toString().toConstChar());
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers printArray
    script_engine.registerFunction("printArray",printArray);

    // calls printArray
    script_engine.compileAndRun(
        "printArray(["
            "0"
            ",10.5"
            ",\"Hello World!\""
        "])");

    return 0;
}
```

Console output:

```
Array contents : [0,10.500000,"Hello World!"]
```

3.1.11. ObjectScriptObject

ObjectScriptObject it defines a array script object.

Constructor

The ObjectScriptObject constructor creates a new object script object. ObjectScriptObject constructor requires a *ScriptEngine* instance.

Syntax

```
ObjectScriptObject(ScriptEngine *_script_engine);
```

Parameters

- *_script_engine* : ScriptEngine instance.

Example

```
#include "zetscript.h"

zetscript::ObjectScriptObject *returnNewObject(
    zetscript::ScriptEngine *_script_engine
){
    // instance new StringScriptObject using ScriptEngine instance
    zetscript::ObjectScriptObject *new_object=new zetscript::ObjectScriptObject(_script_engine);

    //...

    return new_object;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers returnNewObject
    script_engine.registerFunction("returnNewObject",returnNewObject);

    // prints the contents of new array object (empty)
    script_engine.compileAndRun("Console::outln(returnNewObject())");

    return 0;
}
```

Console output:

```
{}
```

Member functions

ObjectScriptObject::elementInstanceOf()

Says whether the element is instance of zetscript data type or a registered type.

Syntax

```
bool    elementInstanceOf<_T>(const String & _key);
```

Template

- *_T* : Zetscript data type or a registered type.

Parameters

- *_key* : The string value as the key.

Return

Returns true if the element at key *_key* it type or extends from type *_T*

Example

```
#include "zetscript.h"

// Check whether the object element is type integer, float or string
void testObjectElementInstanceOf(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ObjectScriptObject *_object
){
    auto keys=_object->getKeys();

    for(int i=0; i < keys.length(); i++){

        printf("Element at key '%s' is type '",keys.get(i).toConstChar());

        if(_object->elementInstanceOf<zetscript::zs_int>(keys.get(i))){
            printf("Integer");
        }

        if(_object->elementInstanceOf<zetscript::zs_float>(keys.get(i))){
            printf("Float");
        }

        if(_object->elementInstanceOf<zetscript::String>(keys.get(i))){
            printf("String");
        }

        printf("'\n");
    }
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers testObjectElementInstanceOf
    script_engine.registerFunction("testObjectElementInstanceOf",testObjectElementInstanceOf);

    // calls printObjectElementTypes
    script_engine.compileAndRun(
        "testObjectElementInstanceOf({"
            "\"key1\":0"                // Element at key 'key1' is type 'Integer'
            ",\"key2\":10.5"             // Element at key 'key2': is type 'Float'
            ",\"key3\":\"Hello World!\"" // Element at key 'key3': is type 'String'
        "})");

    return 0;
}
```

Console output:

```
Element at key 'key1' is type 'Integer'
Element at key 'key2' is type 'Float'
Element at key 'key3' is type 'String'
```

ObjectScriptObject::get()

Returns element of type *_T* at key *_key*.

Syntax

```
_T  get(const String & _key);
```

Template

- *_T* : Zetscript data type or a registered type.

Parameters

- *_key* : The string value as the key.

Return

The element at the specified key in the object.

Example

```
#include "zetscript.h"

// Print contents of an object.
void printObject(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ObjectScriptObject *_object
){

    zetscript::Vector<zetscript::String> keys=_object->getKeys();

    printf("Object contents : {");
    for(int i=0; i < keys.length(); i++){
        if(i>0){
            printf(",");
        }
        if(_object->elementInstanceOf<zetscript::zs_int>(keys.get(i))){
            printf("%s:%i",keys.get(i).toConstChar(),(int)_object->get<zetscript::zs_int>(keys.get(i)));
        }else if(_object->elementInstanceOf<zetscript::zs_float>(keys.get(i))){
            printf("%s:%f",keys.get(i).toConstChar(),_object->get<zetscript::zs_float>(keys.get(i)));
        }else if(_object->elementInstanceOf<zetscript::String>(keys.get(i))){
            printf("%s:'%s'",keys.get(i).toConstChar(),_object->get<zetscript::String>(keys.get(i)).toConstChar());
        }else{
            printf("N/A");
        }
    }
    printf("}\n");
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers 'printObject' function
    script_engine.registerFunction("printObject",printObject);

    // calls printObject
    script_engine.compileAndRun("printObject({i:0,f:10.5,s:\"Hello World\"})");

    return 0;
}
```

Console output:

```
Object contents : {i:0,f:10.500000,s:'Hello World'}
```

ObjectScriptObject::getKeys()

Returns a vector of strings as the keys of the Object.

Syntax

```
Vector<String> getKeys();
```

Parameters

None

Return

A vector of strings as the keys of the Object.

Example

```
#include "zetscript.h"

void printKeys(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ObjectScriptObject *_object
){

    // instance new ObjectScriptObject using ScriptEngine instance
    auto keys=_object->getKeys();
    printf("keys : [");
    for(int i=0; i < keys.length(); i++){
        if(i>0){
            printf(",");
        }
        printf("\"%s\"",keys.get(i).toConstChar());
    }
    printf("]\n");
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers printKeys
    script_engine.registerFunction("printKeys",printKeys);

    script_engine.compileAndRun(
        "var object={"
        "    \"key1\":0"
        "    ,\"key2\":\"hello\""
        "    ,\"key3\":10.0"
        "    ,\"key4\":0"
        "    ,\"key5\":1"
        "    ,\"key6\":2"
        "    ,\"key7\":\"world\"}\n"
        "printKeys(object)\n"

    );

    return 0;
}
```

Console output:

```
keys : ["key1","key2","key3","key4","key5","key6","key7"]
```

ObjectScriptObject::set()

Replaces current value at key *_key* by *_value* of type *_T*.

Syntax

```
void ObjectScriptObject::set(int _key, _T _element)
```

Template

- *_T* : Zetscript data type or a registered type.

Parameters

- *_key* : The string value as the key.
- *_element* : The element to replace.

Return

None

Example

```
#include "zetscript.h"

void modifyObject(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ObjectScriptObject *_object
){

    auto keys=_object->getKeys();
    for(int i=0; i < keys.length(); i++){
        switch(i%3){
        case 0: // set a integer
            _object->set<zetscript::zs_int>(keys.get(i),i);
            break;
        case 1: // set a random float
            _object->set<zetscript::zs_float>(keys.get(i),i*10.2);
            break;
        case 2: // set new string
            _object->set<zetscript::StringScriptObject *>(keys.get(i),new zetscript::StringScriptObject(_script_engine,"Hello"));
            break;

        }
    }
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers modifyObject
    script_engine.registerFunction("modifyObject",modifyObject);

    // create a initialized array, it prints. then prints the
    // the array after call 'modifyObject'
    script_engine.compileAndRun(
        "var o={"
        "   \"key1\":0"
        "   ,\"key2\":\"hello\""
        "   ,\"key3\":10.0"
        "   ,\"key4\":0"
        "   ,\"key5\":1"
        "   ,\"key6\":2"
        "   ,\"key7\":\"world\"}\n"
        "Console::outln(\"Before call 'modifyObject':{0}\",o);"
        "modifyObject(o)\n"
        "Console::outln(\"After call 'modifyObject':{0}\",o);"
    );

    return 0;
}
```

Console output:

```
Before call 'modifyObject':{"key1":0,"key2":"hello","key3":10.000000,"key4":0,"key5":1,"key6":2,"key7":"world"}
```

```
After call 'modifyObject':{"key1":0,"key2":10.200000,"key3":"Hello","key4":3,"key5":40.800000,"key6":"Hello","key7":6}
```

ObjectScriptObject::toString()

Returns a string as the object contents in json format.

Syntax

```
String toString();
```

Parameters

None

Return

A string of the object contents in json format.

Example

```
#include "zetscript.h"

// print array contents
void printObject(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ObjectScriptObject *_object
){
    printf("ScriptObject contents : %s",_object->toString().toConstChar());
}

int main(){
    zetscript::ScriptEngine script_engine;

    // registers printObject
    script_engine.registerFunction("printObject",printObject);

    // calls printObject
    script_engine.compileAndRun(
        "printObject({"
            "\"key1\":0"
            ",\"key2\":10.5"
            ",\"key3\":\"Hello World!\""
        "})");

    return 0;
}
```

Console output:

```
ScriptObject contents : {"key1":0,"key2":10.500000,"key3":"Hello World!"}
```

3.1.12. zetscript::ClassScriptObject

ClassScriptObject it a subclass of *ObjectScriptObject* that implements the *class* type defined in ZetScript. Also it is used as a wrapper to instance a registered type.

3.1.13. zetscript::ScriptFunction

ScriptFunction is a script function type object. *ScriptFunction* type contains function information.

- *id*: Function id
- *name*: function name
- *return_script_type_id*: return script type id.
- *owner_script_type_id*: script type id which script function it belongs
- *properties*: Script function properties

`Value`	Description
`SCRIPT_FUNCTION_PROPERTY_NATIVE_OBJECT_REF`	Tells that the script function it represents a registered native function
`SCRIPT_FUNCTION_PROPERTY_STATIC`	Tells that the script function is static
`SCRIPT_FUNCTION_PROPERTY_MEMBER_FUNCTION`	Tells that script function is a member of a script type
`SCRIPT_FUNCTION_PROPERTY_DEDUCE_AT_RUNTIME`	Tells that the symbol of script function is not resolved yet and it will be resolved at runtime. This situation happens only on registered functions when it registers two or more functions with the same name but different number of parameter or diferent parameter types

3.1.14. zetscript::ScriptEngine

ScriptEngine is the main type of ZetScript in order to register function, register types, compile, execute scripts and much more.

ScriptEngine::ScriptEngine()

The ScriptEngine constructor creates a script engine

Syntax

```
ScriptEngine(int _stack_size);
```

Parameters

- *_stack_size*: Set stack size used by the virtual machine (default 256)

Example

The following example it binds a ZetScript function *sayHelloWorld*.

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine(1024); // It reserves 1K for stack size

    // ...

    return 0;
}
```

ScriptEngine::bindScriptFunction()

Binds script function to be callable from C++.

Syntax

```
std::function<F> bindScriptFunction(const String & _script_function_name);

std::function<F> bindScriptFunction(MemberFunctionScriptObject *_member_function_script_object);

std::function<F> bindScriptFunction(ScriptFunction *_script_function, ScriptObject *_object);

std::function<F> bindScriptFunction(ScriptFunction *_script_function);
```

Template

- *F* : The template C function with the following signature,

```
ReturnType (ParamType1 *, ParamType2 *, ..., ParamType9 *)
```

Where,

ReturnType it can be a registered type or one of the following types,

ReturnType	Description
void	Returns nothing
bool	Returns Boolean
zetscript::zs_int	Returns Integer
zetscript::zs_float	Returns Float
zetscript::String	Returns String
zetscript::StringScriptObject *	Returns String
zetscript::ArrayScriptObject *	Returns Array
zetscript::ObjectScriptObject *	Returns Object
zetscript::ClassScriptObject *	Returns a registered type wrapped in zetscript::ClassScriptObject
T *	Returns a pointer of a register type T

As a parameters, a maximmum of 9 parameters type *ParamType* as a pointer of registered native type or one of the following types,

ParamType	Description
bool *	Calling function in script must pass Boolean type
Calling function in script must pass zetscript::zs_int *	Calling function in script must pass Integer type
zetscript::zs_float *	Calling function in script must pass Float type
zetscript::String *	Calling function in script must pass String type
const char *	Calling function in script must pass String
zetscript::StringScriptObject *	Calling function in script must pass String type
zetscript::ArrayScriptObject *	Calling function in script must pass Array type
zetscript::ObjectScriptObject *	Calling function in script must pass Object type
T *	Calling function in script must pass an instance of a register type T

Parameters

- *_script_function_name*: Bind script function by script function name to bind
- *_member_function_script_object*: Script member function instance
- *_object*: Script object instance owner of the script member function
- *_script_function*: Script function instance

Returns

Returns a *std::function* with the signature provided.

Example

The following example it binds a ZetScript function *sayHelloWorld*.

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine;

    // Compiles 'printHelloWorld' function
    script_engine.compile("function printHelloWorld(){\n"
        "Console::outln(\"Hello world!!\");\n"
    "}");

    // binds 'printHelloWorld'
    auto print_hello_world=script_engine.bindScriptFunction<void()>("printHelloWorld");

    // invoke 'printHelloWorld'
    print_hello_world();

    return 0;
}
```

Console output:

```
Hello world!!
```

ScriptEngine::clear()

It clears all symbols, types, variables from last saved stated.

Syntax

```
void clear();
```

Parameters

None

Returns

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine;

    script_engine.compileAndRun("var i=10;");

    // Prints the value of symbol 'i'
    script_engine.compileAndRun("Console::outln(\"i => \"+i);");

    // Clears current state. After clear, symbol 'i' doesn't exist anymore
    script_engine.clear();

    // throws an error that symbol 'i' does not exist
    try{
        script_engine.compileAndRun("Console::outln(\"i => \"+i);");
    }catch(zetscript::Exception & ex){
        printf("Error: %s\n",ex.what());
    }

    return 0;
}
```

Console output:

```
i => 10
Error: Symbol 'i' not defined
```

ScriptEngine::compile()

Compiles a ZetScript expression. This operation doesn't removes generated code, variables, functions,etc from last compiled code. Each compilation preserves last compiled code. If case to have a reset of everything use *ScriptEngine::clear*

Syntax

```
void    compile(const String & _expresion,const char *__invoke_file__="", int __invoke_line__=-1);
```

Parameters

- *_expresion*: The string as expression to compile
- *__invoke_file__*: The file where compile was invoked. This is used to have some traceability on the source when some error happens (optional)
- *__invoke_line__*: The line where compile was invoked. This is used to have some traceability on the source when some error happens (optional)

Returns

None

Example

```
#include "zetscript.h"
int main(){

    zetscript::ScriptEngine script_engine;

    // Compiles function "add"
    script_engine.compile(
        "function add(_o1,_o2){\n"
        "   return _o1+_o2;\n"
        "}\n"
    );

    // Compiles code that uses "adds" function previusly defined
    script_engine.compile(
        "var b=add(2,3)\n"
        "Console::outln(\"b = \"+b)\n"
    );

    // It runs the code compiled (Prints result of variable 'b')
    script_engine.run();

    return 0;
}
```

Console output:

```
b = 5
```

ScriptEngine::compileAndRun()

Compiles a ZetScript expression and runs generated code. This operation doesn't removes generated code, variables, functions,etc from last compiled code. Each compilation preserves last compiled code. If case to have a reset of everything use *ScriptEngine::clear*

Syntax

```
StackElement    compileAndRun(const String & _expresion, const char *__invoke_file__="", int __invoke_line__=-1);
```

Parameters

- *_expresion*: The string as expression to compile
- *__invoke_file__*: The file where compile was invoked. This is used to have some traceability on the source when some error happens (optional)
- *__invoke_line__*: The line where compile was invoked. This is used to have some traceability on the source when some error happens (optional)

Returns

<<api_data_types.adoc#_zetscriptstackelement[StackElement] as a result of the value returned

Example

```
#include "zetscript.h"
int main(){

    zetscript::ScriptEngine script_engine;

    script_engine.compileAndRun(
        "Console::outln(\"Hello world\")"
    );

    return 0;
}
```

Console output:

```
Hello world
```

ScriptEngine::compileFile()

It compiles a ZetScript file. This operation doesn't removes generated code, variables, functions,etc from last compiled code. Each compilation preserves last compiled code. If case to have a reset of everything use *ScriptEngine::clear*

Syntax

```
void    compileFile(const String & _filename, CompileData *_compile_data=NULL, const char *__invoke_file__="", int __invoke_line__=-1);
```

Parameters

- *_filename*: The file to compile.
- *__invoke_file__*: The file where compile was invoked. This is used to have some traceability on the source when some error happens (optional)
- *__invoke_line__*: The line where compile was invoked. This is used to have some traceability on the source when some error happens (optional)

Returns

None

Example

```
#include "zetscript.h"
int main(){
    zetscript::ScriptEngine script_engine;

    script_engine.compileFile("file.zs");

    return 0;
}
```

The source of file.zs is,

```
function add(_a,_b){
    return _a+_b;
}
var sum=add(10,5);
Console::outln("result : "+sum)
```

Console output:

```

```

ScriptEngine::compileFileAndRun()

It compiles a ZetScript file and runs current generated code. This operation doesn't removes generated code, variables, functions,etc from last compiled code. Each compilation preserves last compiled code. If case to have a reset of everything use *ScriptEngine::clear*

Syntax

```
StackElement    compileFileAndRun(const String & _filename, const char *__invoke_file__="", int __invoke_line__=-1);
```

Parameters

- *_filename*: The file to compile and run.
- *__invoke_file__*: The file where compile was invoked. This is used to have some traceability on the source when some error happens (optional)
- *__invoke_line__*: The line where compile was invoked. This is used to have some traceability on the source when some error happens (optional)

Returns

[_zetscriptstackelement[StackElement]] as a result of the value returned

Example

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine;

    script_engine.compileFileAndRun("file.zs");

    return 0;
}
```

The source of 'file.zs' is,

```
function add(_a,_b){
    return _a+_b;
}
var sum=add(10,5);
Console::outln("result : "+sum)
```

Console output:

```
result : 15
```

ScriptEngine::extends()

Inherits methods from other registered type

Syntax

```
void extends<T,B>()
```

Template parameters

- *T*: The type to be extended to.
- *B*: The type to be extended from.

Note : The method do not supports (by now) a forward register to an extended types when any member o property is being registered on *B*. So, it is important to register **FIRST** all members and properties of *B* **BEFORE** before apply the extension to *T*.

Example

```
#include "zetscript.h"

class MyCppType{
};

class MyCppTypeExtend:public MyCppType{
};


MyCppTypeExtend * MyCppTypeExtend_new(
    zetscript::ScriptEngine *_script_engine
){
    return new MyCppTypeExtend();
}

void MyCppTypeExtend_delete(
    zetscript::ScriptEngine *_script_engine
    , MyCppTypeExtend *_this
){
    delete _this;
}

int main(){
    zetscript::ScriptEngine script_engine; // instance zetscript

    // Register MyCppType as MyCppType in script side
    script_engine.registerType<MyCppType>("MyCppType");

    // Register MyCppTypeExtend as MyCppTypeExtend in script side as instantiable
    script_engine.registerType<MyCppTypeExtend>("MyCppTypeExtend",MyCppTypeExtend_new,MyCppTypeExtend_delete);

    // Tells MyCppTypeExtends extends from MyCppType
    script_engine.extends< MyCppTypeExtend,MyCppType >();

    // Print typeof 'script_my_cpp_type_extend'
    script_engine.compileAndRun(
        "var script_my_cpp_type_extend=new MyCppTypeExtend();\n"
        "Console::outln(\"tyoeof(object) => \"+typeof script_my_cpp_type_extend);\n"
    );
    return 0;
}
```

Console output:

```
tyoeof(object) => type@MyCppTypeExtend
```

ScriptEngine::printGeneratedCode()

Prints information about generated code of the main and other function, script functions.

Header

The header describes the following,

- *Function*: The function name. The main function is described as @MainFunction.
- *Stack code*: Stack required for code
- *Stack local vars*: Stack required for local variables
- *Total stack required*: Total stack required
- *Scopes*: Total scopes

Instructions

Below the header it appears a list of the instructions of generated code with the following columns,

- *NUM*: Instruction number
- *RS*: Required Stack for current instruction
- *AS*: Acumulated Stack in the current instruction
- *BYTE CODE*: Description of the Byte code name used and operands

The meaning of each byte code is described in the following table,

ByteCode	Description
`EQU`	Logical equal
`NOT_EQU`	Logical not equal
`LT`	Less than
`LTE`	Less than equal
`GT`	Greater than
`GTE`	Greater than equal
`LOGIC_AND`	Login AND
`LOGIC_OR`	Logic OR
`INSTANCEOF`	Instanceof
`NOT`	Not
`NEG`	Negate
`ADD`	Addition
`SUB`	Substraction
`DIV`	Division
`MUL`	Multiplication
`MOD`	Modulus
`AND`	Bitwise AND
`OR`	Bitwise OR
`XOR`	Bitwise XOR
`SHL`	Bitwise SHIFT LEFT
`SHR`	Bitwise SHIFT RIGHT
`BWC`	Bitwise complement
`ASSIGN`	Assign
`ADD_ASSIGN`	Addition and assignment
`SUB_ASSIGN`	Substraction and assignment
`MUL_ASSIGN`	Multiplication and assignment
`DIV_ASSIGN`	Division and assignment

ByteCode	Description
MOD_ASSIGN	Modulus and assignment
AND_ASSIGN	Bitwise AND and assignment
OR_ASSIGN	Bitwise OR and assignment
XOR_ASSIGN	Bitwise XOR and assignment
SHL_ASSIGN	Bitwise SHIFT LEFT and assignment
SHR_ASSIGN	Bitwise SHIFT RIGHT and assignment
PUSH_STK_GLOBAL_IRGO	Push global variable from return function to stack
PUSH_STK_GLOBAL	Push global variable to stack
PUSH_STK_LOCAL	Push local variable to stack
PUSH_STK_REF	Push argument reference to stack
PUSH_STK_THIS	Push current object instance to stack
PUSH_STK_ARRAY@ITEM	Push array item to stack
PUSH_STK_OBJ@ITEM	Push object item to stack
PUSH_STK_THIS@VAR	Push member variable to stack
LOAD_GLOBAL	Load global variable to stack
LOAD_LOCAL	Load local variable to stack
LOAD_REF	Load argument reference to stack
LOAD_THIS	Load current instance to stack
LOAD_CONSTRUCTOR_FUNCT	Load constructor function to stack
LOAD_???	Unresolved load symbol. It tries to resolve and load at runtime
CALL	Performs a direct call
CALL???	Unresolved direct call. It tries to load at runtime
STK_CALL	Performs a call from current stack
MEMBER_CALL	Performs member call
LOAD_THIS@VAR	Load member variable to stack
LOAD_THIS@FUN	Load member function to stack
LOAD_ARRAY@ITEM	Load array item to stack
LOAD_OBJ@ITEM	Load object item to stack
LOAD_FUN	Load function to stack
LOAD_UNDEFINED	Load undefined value to stack
LOAD_NULL	Load Null value to stack
LOAD_STK	Load stack element to stack
LOAD_STR	Load *String* value to stack
LOAD_FLT	Load *Float* value to stack
LOAD_BOOL	Load *Boolean* value to stack
LOAD_INT	Load *Integer* value to stack
LOAD_TYPE	Load *Type* value to stack
JMP	Performs an unconditional jump.
JMP_CASE	Performs a jump Switch-Case.
JNT	Performs a jump if not true
JT	Performs jump if true.
JE_CASE	Performs a jump if equal Switch-Case
CALL_CONSTRUCTOR	Performs a constructor call

ByteCode	Description
NEW_ARRAY	Creates an array object
PUSH_AITEM	Push array item to stack
RET	Return
NEW_OBJECT_BY_TYPE	Creates new object by type
NEW_OBJECT_BY_VALUE	Creates new object by value
DELETE	Deletes object
POP_SCOPE	Pop scope
PUSH_SCOPE	Push scope
PUSH_OITEM	Push object item
NEW_OBJECT	Creates new object
NEW_STR	Creates new string
IT_INIT	Performs iterator init
STORE_CONST	Performs a constant assignment
PRE_INC	Performs a preincrement
PRE_DEC	Performs a predecrement
POST_INC	Performs a postincrement
POST_DEC	Performs a postdecrement
RESET_STACK	Resets stack
TYPEOF	Typeof
IN	In operator

Syntax

```
void printGeneratedCode()
```

Example

```
#include "zetscript.h"
int main(){

    zetscript::ScriptEngine script_engine;

    script_engine.compile(
        "function sum(_a,_b){\n"
        " return _a+_b\n"
        "}\n"
        "\n"
        "Console::outln(\"result: {0}\",sum(5+10))\n"
    );

    script_engine.printGeneratedCode();
}
```

Console output:

```
-----------------------------------------------------------------

 ScriptFunction: '@MainFunction'
 Stack code: 2
 Stack local vars: 0
 Total stack required: 2
 Scopes: 2

 NUM |RS|AS|               BYTE CODE
-----+--+--+-------------------------------------------------
[0000| 1|01]    LOAD_STRING     "result: {0}"
[0001| 1|02]    LOAD_INT        15
[0002| 0|02]    CALL            sum arg:1 ret:1
[0003|-1|00]    CALL            Console::outln  arg:2 ret:1 [RST]
```

```
------------------------------------------------------------

 ScriptFunction: 'sum'
 Stack code: 1
 Stack local vars: 2
 Total stack required: 3
 Scopes: 0

 NUM |RS|AS|               BYTE CODE
-----+--+--+-------------------------------------------------
[0000| 1|01]    ADD         Local['_a'],Local['_b']
[0001| 0|00]    RET
```

ScriptEngine::pushStackElement()

Pushes stack element to stack to be readed later as a return function in ZetScript. The order how stack elements are pushed/popped is FIFO (First into stack First out to be processed). As a difference from register normal function seen in ScriptEngine::registerFunction this operation allows return multiple values.

Syntax

```
void           pushStackElement(
    zetscript::StackElement _stk
);
```

Example

```
#include "zetscript.h"

void returnStackElement(zetscript::ScriptEngine *_script_engine){

    // Converts primitive type to stack element
    zetscript::StackElement r1=_script_engine->toStackElement<zetscript::zs_int>(10);
    zetscript::StackElement r2=_script_engine->toStackElement<zetscript::zs_float>(10.5);
    zetscript::StackElement r3=_script_engine->toStackElement<zetscript::StringScriptObject *>(
        new zetscript::StringScriptObject(_script_engine,"Hello World!!!")
    );

    // Push stack stack elements FIFO order (First into stack First out to be readed in ZetScript)
    _script_engine->pushStackElement(r1);
    _script_engine->pushStackElement(r2);
    _script_engine->pushStackElement(r3);
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers the C function
    script_engine.registerFunction("returnStackElement",returnStackElement);

    // // Compiles and runs a script
    script_engine.compileAndRun(
        "var r1,r2,r3;\n"
        "r3,r2,r1 = returnStackElement();\n"
        "Console::outln(\"r1:{0} r2:{1} r3:{2}\",r1,r2,r3);"
    );
    return 0;
}
```

Console output:

```
r1:10 r2:10.500000 r3:Hello World!!!
```

ScriptEngine::registerConstantBoolean()

Registers constant Boolean value.

Syntax

```
void registerConstantBoolean(const String & _const_name, bool _value, const char *registered_file="", short registered_line=-1);
```

Function parameters

- *_const_name* : Constant name
- *_value* : Boolean value
- *_registered_file* : Source file where the function was registered
- *_registered_line* : Line file where the function was registered

Returns

None

Example

```
#include "zetscript.h"

int main(){

    // instance ScriptEngine
    zetscript::ScriptEngine script_engine;

    // Registers a constant Boolean
    script_engine.registerConstantBoolean("MY_CONSTANT_BOOL",true);

    // Prints the value of each registered constant
    script_engine.compileAndRun(
        "Console::outln(\"MY_CONSTANT_BOOL:\"+MY_CONSTANT_BOOL);\n"
    );
}
```

Console output:

```
MY_CONSTANT_BOOL:true
```

ScriptEngine::registerConstantFloat()

Registers constant Float value.

Syntax

```
void registerConstantFloat(const String & _const_name, zs_float _value, const char *registered_file="", short registered_line=-1);
```

Function parameters

- *_const_name* : Constant name
- *_value* : Float value
- *_registered_file* : Source file where the function was registered
- *_registered_line* : Line file where the function was registered

Returns

None

Example

```
#include "zetscript.h"

int main(){

    // instance ScriptEngine
    zetscript::ScriptEngine script_engine;

    // Registers a constant Float
    script_engine.registerConstantFloat("MY_CONSTANT_FLOAT",2.5e-3);

    // Prints the value of each registered constant
    script_engine.compileAndRun(
        "Console::outln(\"MY_CONSTANT_FLOAT:\"+MY_CONSTANT_FLOAT);\n"
    );
}
```

Console output:

```
MY_CONSTANT_FLOAT:0.002500
```

ScriptEngine::registerConstantInteger()

Registers constant Integer value.

Syntax

```
void registerConstantInteger(const String & _const_name, zs_int _value, const char *registered_file="", short registered_line=-1);
```

Function parameters

- *_const_name* : Constant name
- *_value* : Integer value
- *_registered_file* : Source file where the function was registered
- *_registered_line* : Line file where the function was registered

Returns

None

Example

```
#include "zetscript.h"

int main(){

    // instance ScriptEngine
    zetscript::ScriptEngine script_engine;

    // Registers a constant Integer
    script_engine.registerConstantInteger("MY_CONSTANT_INT",10);

    // Prints the value of each registered constant
    script_engine.compileAndRun(
        "Console::outln(\"MY_CONSTANT_INT:\"+MY_CONSTANT_INT);\n"
    );
}
```

Console output:

```
MY_CONSTANT_INT:10
```

ScriptEngine::registerConstantString()

Registers constant String value.

Syntax

```
void registerConstantString(const String & _const_name, const String & _value, const char *registered_file="", short registered_line=-1);
```

Function parameters

- *_const_name* : Constant name
- *_value* : String value
- *_registered_file* : Source file where the function was registered
- *_registered_line* : Line file where the function was registered

Returns

None

Example

```
#include "zetscript.h"

int main(){

    // instance ScriptEngine
    zetscript::ScriptEngine script_engine;

    // Registers a constant String
    script_engine.registerConstantString("MY_CONSTANT_STR","my_string");

    // Prints the value of each registered constant
    script_engine.compileAndRun(
        "Console::outln(\"MY_CONSTANT_STR:\"+MY_CONSTANT_STR);\n"
    );
}
```

Console output:

```
MY_CONSTANT_STR:my_string
```

ScriptEngine::registerConstMemberProperty

Registers a C function as a constant member property of a registered type.

Syntax

```
void    registerConstMemberProperty<T>(
        const zetscript::String & _property_name
        ,F _c_function
        , const char *_registered_file=""
        ,short _registered_line=-1
);
```

Template parameters

- *T*: The registered type

Function parameters

- *_property_name* : Property name
- *_c_function*: A C function that returns a type as a value of the property with the following signature,

```
ReturnType (ScriptEngine *_script_engine){
    ....
}
```

Where,

ReturnType it can be a registered type or one of the following types,

ReturnType	Description
bool	Returns Boolean
zetscript::zs_int	Returns Integer
zetscript::zs_float	Returns Float
zetscript::String	Returns String
zetscript::StringScriptObject] *	Returns String
zetscript::ArrayScriptObject *	Returns Array
zetscript::ObjectScriptObject *	Returns Object
zetscript::ClassScriptObject *	Returns a registered type wrapped in zetscript::ClassScriptObject

The function always includes *zetscript::ScriptEngine* * as FIRST parameter.

- *_registered_file* : Source file where the function was registered
- *_registered_line* : Line file where the function was registered

Returns

None

Example

```
#include "zetscript.h"

class Number{
//...
};

zetscript::zs_float NumberZs_MAX_VALUE(
    zetscript::ScriptEngine *_script_engine
){
    ZS_UNUSUED_PARAM(_script_engine);
    return FLT_MAX;
}


int main(){
    zetscript::ScriptEngine script_engine;
```

```
    script_engine.registerType<Number>("Number");

    script_engine.registerConstMemberProperty<Number>("MAX_VALUE",NumberZs_MAX_VALUE);

    script_engine.compileAndRun(

        "Console::outln(\"Number::MAX_VALUE => {0}\",Number::MAX_VALUE);"
    );

    return 0;
}
```

Console output:

```
Number::MAX_VALUE => 340282346638528859811704183484516925440.000000
```

ScriptEngine::registerConstructor()

Registers C function as a constructor of a registered type.

Syntax

```
void registerConstructor<T>( F _c_function);
```

Types

- *T*: The registered type

Parameters

- *_c_function*: C function reference to be registered

Returns

None

Example

```
#include "zetscript.h"

// C++ class to be registered
class Number{
public:
    float value;

    Number(){
        value=0;
    }
};

// defines new function for Number object
Number *NumberZs_new(
    zetscript::ScriptEngine *_script_engine
){
    return new  Number();
}

// defines new function for Number constructor
void NumberZs_constructor(
    zetscript::ScriptEngine *_script_engine
    ,Number *_this
    ,zetscript::zs_float * _value
){
    printf("Constructor value : %.02f\n",*_value);
    _this->value=*_value;
}

// defines delete function for Number object
void NumberZs_delete(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    delete _this;
}


int main(){
    zetscript::ScriptEngine script_engine;

    // Register class Number as instanciable
    script_engine.registerType<Number>("Number",NumberZs_new,NumberZs_delete);

    // Register class Number constructor
    script_engine.registerConstructor<Number>(NumberZs_constructor);

    // Prints the result of the evaluated script by console
    script_engine.compileAndRun(
        "var n=new Number(20.5);"
    );

    return 0;
}
```

Console output:

```
Constructor value : 20.50
```

ScriptEngine::registerFunction()

Registers a C function to be callable from ZetScript.

Syntax

```
void registerFunction(
    const zetscript::String & _function_name
    ,F _c_function
    ,const char *_registered_file=""
    ,short _registered_line=-1
);
```

Parameters

- *_function_name* : Function name.
- *_c_function* : A C function to register with the following signature,

```
ReturnType (ScriptEngine *_script_engine, ParamType *_arg1, ParamType *_arg2, ..., ParamType *_arg9 )
```

Where,

ReturnType it can be a registered type or one of the following types,

ReturnType	Description
void	Returns nothing
bool	Returns Boolean
zetscript::zs_int	Returns Integer
zetscript::zs_float	Returns Float
zetscript::String	Returns String
zetscript::StringScriptObject] *	Returns String
zetscript::ArrayScriptObject *	Returns Array
zetscript::ObjectScriptObject *	Returns Object
zetscript::ClassScriptObject *	Returns a registered type wrapped in zetscript::ClassScriptObject
T *	Returns a pointer of a register type T

As a parameters, the function must always include *zetscript::ScriptEngine* * as the FIRST parameter and from SECOND till a maximmum of 9 parameters type *ParamType* as one of the following types,

ParamType	Description
bool *	Calling function in script must pass Boolean type
Calling function in script must pass zetscript::zs_int *	Calling function in script must pass Integer type
zetscript::zs_float *	Calling function in script must pass Float type
zetscript::String *	Calling function in script must pass String type
const char *	Calling function in script must pass String
zetscript::StringScriptObject] *	Calling function in script must pass String type
zetscript::ArrayScriptObject *	Calling function in script must pass Array type
zetscript::ObjectScriptObject *	Calling function in script must pass Object type
T *	Calling function in script must pass an instance of a register type T
zetscript::StackElement *	This paramater is declared as generic and it can pass **ANY** type from ZetScript. In the C function, StackElement has to be deduced manually

- *_registered_file* : Source file where the function was registered
- *_registered_line* : Line file where the function was registered

Returns

None

Example

The following example registers a C function *sayHelloWorld* and it is called in ZetScript.

```
#include "zetscript.h"

// ScriptEngine C++ interface function
void sayHelloWorld(zetscript::ScriptEngine *_script_engine){
    printf("Hello world\n");
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers sayHelloWorld as 'sayHelloWorld' symbol name
    script_engine.registerFunction("sayHelloWorld",sayHelloWorld);

    // Evaluates a script where it calls 'sayHelloWorld' function
    script_engine.compileAndRun(
        "sayHelloWorld();"
    );

    return 0;
}
```

Console output:

```
Hello world
```

To see more and complete examples see [_call_c_from_zetscript[Call C++ from ZetScript]]

ScriptEngine::registerMemberFunction()

Registers a C function as a member function of a registered type.

Syntax

```
void registerMemberFunction<T>(
    const zetscript::String & _member_function_name
    ,F _c_function
    ,const char *_registered_file=""
    ,short _registered_line=-1
);
```

Template parameters

- *T*: The registered type

Function parameters

- *_member_function_name* : Member function name
- *_c_function*: A C function with the following signature,

```
ReturnType (ScriptEngine *_script_engine, T *_this, ParamType *_arg1, ..., ParamType *_arg8 ){
    ....
}
```

Where,

ReturnType it can be a registered type or one of the following types,

ReturnType	Description
void	Returns nothing
bool	Returns Boolean
zetscript::zs_int	Returns Integer
zetscript::zs_float	Returns Float
zetscript::String	Returns String
zetscript::StringScriptObject] *	Returns String
zetscript::ArrayScriptObject *	Returns Array
zetscript::ObjectScriptObject *	Returns Object
zetscript::ClassScriptObject *	Returns a registered type wrapped in zetscript::ClassScriptObject

As parameters, the function must always include *zetscript::ScriptEngine* * as the FIRST parameter,the current instance as the SECOND parameter and the THIRD till a maximmum of 8 parameters type *ParamType* as a pointer of registered native type or one of the following types,

ParamType	Description
bool *	Calling function in script must pass Boolean type
Calling function in script must pass zetscript::zs_int *	Calling function in script must pass Integer type
zetscript::zs_float *	Calling function in script must pass Float type
zetscript::String *	Calling function in script must pass String type
const char *	Calling function in script must pass String
zetscript::StringScriptObject] *	Calling function in script must pass String type
zetscript::ArrayScriptObject *	Calling function in script must pass Array type
zetscript::ObjectScriptObject *	Calling function in script must pass Object type
T *	Calling function in script must pass an instance of a register type T
zetscript::StackElement *	This paramater is declared as generic and it can pass **ANY** type from ZetScript. In the C function, StackElement has to be deduced manually

- *_registered_file* : Source file where the function was registered

- *_registered_line* : Line file where the function was registered

Returns

None

Example

```
#include "zetscript.h"

// C++ class to be registered
class Number{
public:
    float value;

    Number(){
        value=0;
    }
};

// defines new function for Number object
Number *NumberZs_new(
    zetscript::ScriptEngine *_script_engine
){
    return new  Number();
}

// defines new function for Number constructor
void NumberZs_constructor(
    zetscript::ScriptEngine *_script_engine
    ,Number *_this
    ,zetscript::zs_float * _value
){
    _this->value=*_value;
}

// defines get negative
zetscript::zs_float NumberZs_getValue(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    return _this->value;
}

// defines delete function for Number object
void NumberZs_delete(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    delete _this;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Register class Number as instanciable
    script_engine.registerType<Number>("Number",NumberZs_new,NumberZs_delete);

    // Register class Number constructor
    script_engine.registerConstructor<Number>(NumberZs_constructor);

    // Register member function Number::getValue
    script_engine.registerMemberFunction<Number>("getValue",NumberZs_getValue);

    // Prints the result of the evaluated script by console
    script_engine.compileAndRun(
        "var n=new Number(20.5);\n"
        "Console::outln(\"n.getValue() : \"+n.getValue())\n"
    );

    return 0;
}
```

Console output:

```
n.getValue() : 20.500000
```

ScriptEngine::registerMemberPropertyMetamethod()

Registers C function as member property metamethod of a registered type

Syntax

```
void    registerMemberPropertyMetamethod<T>(
    const zetscript::String & _property_name
    ,const zetscript::String & _metamethod_name
    ,F _c_function
    ,const char *_registered_file=""
    ,short _registered_line=-1
);
```

Template parameters

- *T*: The registered type

Function parameters

- *_property_name* : Property name has to be one of the properties seen on Properties - Member functions
- *_metamethod_name* : One of the following metamethod names,
- *_c_function*: A C function.

For setters (i.e _set,_addassign,etc) the following signature,

```
ReturnType (ScriptEngine *_script_engine, T *_this, ParamType *_arg1){
    ....
}
```

For getters, post and pre increment/decrements the following signature,

```
ReturnType (ScriptEngine *_script_engine, T *_this){
    ....
}
```

Where,

ReturnType it can be a registered type or one of the following types,

ReturnType	Description
void	Returns nothing
bool	Returns Boolean
zetscript::zs_int	Returns Integer
zetscript::zs_float	Returns Float
zetscript::String	Returns String
zetscript::StringScriptObject *	Returns String
zetscript::ArrayScriptObject *	Returns Array
zetscript::ObjectScriptObject *	Returns Object
zetscript::ClassScriptObject *	Returns a registered type wrapped in zetscript::ClassScriptObject

As parameters, the function must always include *zetscript::ScriptEngine* * as the FIRST parameter,the current instance as the SECOND parameter. *ParamType* it has to be one of the following types,

ParamType	Description
bool *	Calling function in script must pass Boolean type
Calling function in script must pass zetscript::zs_int *	Calling function in script must pass Integer type
zetscript::zs_float *	Calling function in script must pass Float type
zetscript::String *	Calling function in script must pass String type
const char *	Calling function in script must pass String

zetscript::StringScriptObject *	Calling function in script must pass String type
zetscript::ArrayScriptObject *	Calling function in script must pass Array type
zetscript::ObjectScriptObject *	Calling function in script must pass Object type
T *	Calling function in script must pass an instance of a register type T
zetscript::StackElement *	This paramater is declared as generic and it can pass **ANY** type from ZetScript. In the C function, StackElement has to be deduced manually

- *_registered_file* : Source file where the function was registered
- *_registered_line* : Line file where the function was registered

Returns

None

Example

```
#include "zetscript.h"

// C++ class to be registered
class Number{
public:
    float value;

    Number(){
        value=0;
    }
};

// defines new function for Number object
Number *NumberZs_new(
    zetscript::ScriptEngine *_script_engine
){
    return new  Number();
}

// defines new function for Number constructor
void NumberZs_constructor(
    zetscript::ScriptEngine *_script_engine
    ,Number *_this
    ,zetscript::zs_float *_value
){
    _this->value=*_value;
}

// defines get negative
zetscript::zs_float NumberZs_get(
    zetscript::ScriptEngine *_script_engine
    ,Number *_this
){
    return _this->value;
}

// defines delete function for Number object
void NumberZs_delete(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    delete _this;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Register class Number as instanciable
    script_engine.registerType<Number>("Number",NumberZs_new,NumberZs_delete);

    // Register class Number constructor
    script_engine.registerConstructor<Number>(NumberZs_constructor);

    // register property getter Number::value
    script_engine.registerMemberPropertyMetamethod<Number>("value","_get",NumberZs_get);

    // Prints the result of the evaluated script by console
    script_engine.compileAndRun(
        "var n=new Number(20.5);\n"
```

```
        "Console::outln(\"n : \"+n)\n"
    );

    return 0;
}
```

Console output:

```
n : {"value":20.500000}
```

ScriptEngine::registerStaticMemberFunction()

Registers a C function as static member function of a registered type.

Syntax

```
void registerStaticMemberFunction<T>(
    const zetscript::String & _static_member_name
    , F _function
);
```

Template parameters

- *T*: The registered type

Function parameters

- *_function_name* : The function name to be refered in ZetScript. A static member function can be defined as metamethod if *_function_name* matches some of the name of the metamethod seen at The language.Class.Member Functions. Static metamethods
- *_c_function* : A C function to register with the following signature,

```
ReturnType (ScriptEngine *_script_engine, ParamType *_arg1, ParamType *_arg2, ..., ParamType *_arg9 )
```

Where,

ReturnType it can be a registered type or one of the following types,

ReturnType	Description
void	Returns nothing
bool	Returns Boolean
zetscript::zs_int	Returns Integer
zetscript::zs_float	Returns Float
zetscript::String	Returns String
zetscript::StringScriptObject] *	Returns String
zetscript::ArrayScriptObject *	Returns Array
zetscript::ObjectScriptObject *	Returns Object
zetscript::ClassScriptObject *	Returns a registered type wrapped in zetscript::ClassScriptObject
T *	Returns a pointer of a register type T

As a parameters, the function must always include *zetscript::ScriptEngine* * as the FIRST parameter and from SECOND till a maximmum of 9 parameters type *ParamType* as one of the following types,

ParamType	Description
bool *	Calling function in script must pass Boolean type
Calling function in script must pass zetscript::zs_int *	Calling function in script must pass Integer type
zetscript::zs_float *	Calling function in script must pass Float type
zetscript::String *	Calling function in script must pass String type
const char *	Calling function in script must pass String
zetscript::StringScriptObject] *	Calling function in script must pass String type
zetscript::ArrayScriptObject *	Calling function in script must pass Array type
zetscript::ObjectScriptObject *	Calling function in script must pass Object type
T *	Calling function in script must pass an instance of a register type T
zetscript::StackElement *	This paramater is declared as generic and it can pass **ANY** type from ZetScript. In the C function, StackElement has to be deduced manually

- *_registered_file* : Source file where the function was registered

- *_registered_line* : Line file where the function was registered

Returns

None

Example

```
#include "zetscript.h"
#include <math.h>

class Number{
    //...
};

zetscript::zs_float NumberZs_pow(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::zs_float *_base
    ,zetscript::zs_float *_exp
){

    return pow(*_base,*_exp);
}


int main(){
    zetscript::ScriptEngine script_engine;

    script_engine.registerType<Number>("Number");

    script_engine.registerStaticMemberFunction<Number>("pow",NumberZs_pow);

    script_engine.compileAndRun(

        "Console::outln(\"Number::pow(2,2) => {0}\",Number::pow(2,2));"
    );

    return 0;
}
```

Console output:

```
Number::pow(2,2) => 4.000000
```

ScriptEngine::registerType()

Registers a type in ZetScript.

Syntax

```
ScriptType * registerType<T>(
    const zetscript::String & _script_type_name
    , F _new_c_function=NULL
    , F _delete_c_function=NULL
);
```

Template parameters

- *T*: The type to register.

Parameters

- *_script_type_name*: The script type name.
- *_new_c_function* (Optional): It defines the c function that creates the native type when an instance of this type is created in ZetScript (Optional if it's instantiable). The signature of the c function is,

```
T * (ScriptEngine *_script_engine);
```

- *_delete_c_function* : It defines the c function that destroys the native type when a instance of this type is destroyed in ZetScript (Optional if it's instantiable). The signature of the c function is,

```
void RegisteredType_delete(ScriptEngine *_script_engine, RegisteredType *_this);
```

There's two ways to register a type,

- Register as non instantiable type
- Register as instantiable type

Register type as non instantiable

Register as non instantiable type it means that it **CANNOT** create or instantiate new objects of that *type* in ZetScript using the *new* operator, so the object only can be acceded by reference returned by some registered function. This kind of register is useful when it wants to have a control of instantiated objects in the native application.

If for example we perform the following code,

```
#include "zetscript.h"
// Defines  MyType
class MyType{
//...
};

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers MyType as not instantiable
    script_engine.registerType<MyType>("MyType");

    // tries to instance t as MyType where MyType is NOT instantiable
    try{
        script_engine.compile("var t=new MyType()");
    }catch(zetscript::Exception & _ex){
        printf("%s\n",_ex.what());
    }

    return 0;
}
```

Will throw the following error,

```
Cannot create object type 'MyType' because it has been defined as not instantiable. To solve this issue, register type 'MyType' as
instantiable (i.e register type 'MyType' with new/delete functions)
```

The only way to operate with a 'MyType' object non instantiable is by a returning a reference of object 'MyType' from registered C function as it shows in the following example,

```
#include "zetscript.h"

// Defines  MyType
class MyType{
//...
};

// defines global variable my_type
MyType *my_type=NULL;

MyType *returnMyType(
    zetscript::ScriptEngine *_script_engine
){
    return my_type;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Instances MyType
    my_type=new MyType();

    // Registers type 'MyType' as not instantiable
    script_engine.registerType<MyType>("MyType");

    // registers returnMyType function
    script_engine.registerFunction("returnMyType",returnMyType);

    // Gets an instance of 'MyType' and stores in 't'
    // Instances an object of type 'MyType' and stores in 't'
    script_engine.compileAndRun(
        "var t=returnMyType();\n"
        "Console::outln(\"typeof t =>\"+(typeof t));"
    );

    delete my_type;

    return 0;
}
```

Console output:

```
typeof t =>type@MyType
```

Register type as instantiable

Register as instantiable type it means that it **CAN** instantiate objects of that 'type' in ZetScript code using operator *new*.

Example

In the following example it will show an example of registered type instantiable,

```
#include "zetscript.h"

// Defines  MyType
class MyType{
//...
};

// defines global variable my_type
MyType *my_type=NULL;

MyType *MyTypeZs_new(
    zetscript::ScriptEngine *_script_engine
){
    return new MyType();
}

void MyTypeZs_delete(
    zetscript::ScriptEngine *_script_engine
    ,MyType *_this
){
    delete _this;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Instances MyType
    my_type=new MyType();

    // Registers type 'MyType' as instantiable
    script_engine.registerType<MyType>("MyType",MyTypeZs_new,MyTypeZs_delete);

    // Instances an object of type 'MyType' and stores in 't'
    script_engine.compileAndRun(
        "var t=new MyType();\n"
        "Console::outln(\"typeof t =>\"+(typeof t));\n"
    );

    delete my_type;

    return 0;
}
```

Console output:

```
typeof t =>type@MyType
```

ScriptEngine::unrefLifetimeObject()

Dereferences a binded script object.

Syntax

```
void unrefLifetimeObject(ScriptObject *_script_object);
```

Parameters

- *_script_object*: Script object to be dereferenced.

Returns

None

Example

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine;

    // Eval an expression that returns a 'Object' type as result
    auto result=script_engine.compileAndRun("return {f1:10,f2:20,f4:40}");

    // Converts stack element to ObjectScriptObject
    zetscript::ObjectScriptObject *result_object=script_engine.stackElementTo<zetscript::ObjectScriptObject *>(result);

    // ...
    // perform some operations with 'result_object'
    // ...

    // unref lifetime object due is not needed anymore.
    script_engine.unrefLifetimeObject(result_object);

    return 0;
}
```

ScriptEngine::saveState()

Saves the current registered symbols as variables, types ot functions

Syntax

```
void saveState();
```

Parameters

None

Returns

None

Example

```
#include "zetscript.h"
int main(){
    zetscript::ScriptEngine script_engine;

    // declares and initializes variable 'i'
    script_engine.compileAndRun("var i=10;");

    // saves current state, so variable 'i' is saved
    script_engine.saveState();

    // Clears current state. It doesn't clear symbol 'i' because it was saved
    script_engine.clear();

    // prints 'i' after clear state
    script_engine.compileAndRun("Console::outln(\"i after clear => \"+i);");

    return 0;
}
```

Console output:

```
i after clear => 10
```

ScriptEngine::stackElementTo()

Binds StackElement to a primitive or registered type

Syntax

```
template<typename C>
C   stackElementTo(StackElement * _stack_element);
```

Template parameters

- *C* : The type to be converted as the one of the following,
  - bool
  - zetscript::zs_int
  - zetscript::zs_float
  - zetscript::String
  - zetscript::StringScriptObject *
  - zetscript::ArrayScriptObject *
  - zetscript::ObjectScriptObject *
  - A pointer of a register type

Parameters

- _stack_element: Stack element to converted from

Returns

A *C* type as a result of conversion

Example

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine;

    // Returns Object in StackElement as a result of compile and run "return {f1:10,f2:20,f4:40}"
    zetscript::StackElement result=script_engine.compileAndRun("return {f1:10,f2:20,f4:40}");

    // Converts stack element to ObjectScriptObject
    zetscript::ObjectScriptObject *result_object=script_engine.stackElementTo<zetscript::ObjectScriptObject *>(result);

    // ...
    // perform some operations with 'result_object'
    // ...

    // unref lifetime object due is not needed anymore.
    script_engine.unrefLifetimeObject(result_object);

    return 0;
}
```

ScriptEngine::stackElementTypeToString()

Returns the type of the StackElement as String

Syntax

```
String  stackElementTypeToString(StackElement * _stack_element);
```

Parameters

- _stack_element: Stack element pointer

Returns

A String as the type

Example

```
#include "zetscript.h"

void printStackElementType(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::StackElement *_stack_element
){
    printf("StackElement type is '%s'\n"
        ,_script_engine->stackElementTypeToString(_stack_element).toConstChar()
    );
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Register C function
    script_engine.registerFunction("printStackElementType",printStackElementType);

    // Compiles and runs script
    script_engine.compileAndRun(
        "printStackElementType(1); // pass Integer value to printStackElementType\n"
        "printStackElementType(5.2); // pass Float value to printStackElementType \n"
        "printStackElementType(\"Hello world!\"); // pass String value to printStackElementType \n"
    );


    return 0;
}
```

Console output:

```
StackElement type is 'Integer'
StackElement type is 'Float'
StackElement type is 'String'
```

ScriptEngine::stackElementValueToString()

Returns the contents StackElement value as String

Syntax

```
String  stackElementTypeToString(StackElement * _stack_element);
```

Parameters

- _stack_element: Stack element pointer

Returns

A String as the StackElement value

Example

```
#include "zetscript.h"

void printStackElementValue(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::StackElement *_stack_element
){
    printf("StackElement value is '%s'\n"
        ,_script_engine->stackElementValueToString(_stack_element).toConstChar()
    );
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Register C function
    script_engine.registerFunction("printStackElementValue",printStackElementValue);

    // Compiles and runs script
    script_engine.compileAndRun(
        "printStackElementValue(1); // pass Integer value to printStackElementValue\n"
        "printStackElementValue(5.2); // pass Float value to printStackElementValue \n"
        "printStackElementValue(\"Hello world!\"); // pass String value to printStackElementValue \n"
    );


    return 0;
}
```

Console output:

```
StackElement value is '1'
StackElement value is '5.200000'
StackElement value is 'Hello world!'
```

ScriptEngine::toStackElement()

Converts primitive or registered type to StackElement.

Syntax

```
template<typename C>
zetscript::StackElement toStackElementTo(C _value);
```

Template parameters

- *C* : The value to be converted as the one of the following,
  - bool
  - zetscript::zs_int
  - zetscript::zs_float
  - zetscript::String
  - zetscript::StringScriptObject *
  - zetscript::ArrayScriptObject *
  - zetscript::ObjectScriptObject *
  - A pointer of a register type

Parameters

- *__value*: Value to be converted from.

Returns

StackElement as a result of conversion

Example

```
#include "zetscript.h"

void returnStackElement(zetscript::ScriptEngine *_script_engine){

    // Converts primitive type to stack element
    zetscript::StackElement r1=_script_engine->toStackElement<zetscript::zs_int>(10);
    zetscript::StackElement r2=_script_engine->toStackElement<zetscript::zs_float>(10.5);
    zetscript::StackElement r3=_script_engine->toStackElement<zetscript::StringScriptObject *>(
        new zetscript::StringScriptObject(_script_engine,"Hello World!!!")
    );

    // Push stack stack elements FIFO order (First into stack First out to be readed in ZetScript)
    _script_engine->pushStackElement(r1);
    _script_engine->pushStackElement(r2);
    _script_engine->pushStackElement(r3);
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers the C function
    script_engine.registerFunction("returnStackElement",returnStackElement);

    // // Compiles and runs a script
    script_engine.compileAndRun(
        "var r1,r2,r3;\n"
        "r3,r2,r1 = returnStackElement();\n"
        "Console::outln(\"r1:{0} r2:{1} r3:{2}\",r1,r2,r3);"
    );
    return 0;
}
```

Console output:

```
r1:10 r2:10.500000 r3:Hello World!!!
```

3.2. Call C++ from ZetScript

This chapter aims to explain how to call C++ from ZetScript through a set examples. The examples shows each type of returning/passing parametres to/from ZetScript by defining a C function that has to be registered as it describes ScriptEngine::registerFunction.

3.2.1. Return types

This section will explain how to register C functions that return the following ZetScript types,

- Boolean
- Integer
- Float
- String
- Array
- Object
- Registered type
- Any

Return Boolean

To register a C function that returns a Boolean it must specify *bool* as return type.

Example

```
#include "zetscript.h"

bool returnBoolean(
    zetscript::ScriptEngine *_script_engine
){
    return true;
}

int main(){

    zetscript::ScriptEngine script_engine;

    // Registers function
    script_engine.registerFunction("returnBoolean",returnBoolean);

    // // Compiles and runs a script
    script_engine.compileAndRun(
        "Console::outln(\"result : \"+returnBoolean());"
    );
    return 0;
}
```

Console output:

```
result : true
```

Return Integer

To register a C function that returns a Integer it must specify *zs_int* as return type.

Example

```
#include "zetscript.h"

zetscript::zs_int returnInteger(
    zetscript::ScriptEngine *_script_engine
){
    return 10;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers the C function
    script_engine.registerFunction("returnInteger",returnInteger);

    // // Compiles and runs a script
    script_engine.compileAndRun(
        "Console::outln(\"result : \"+returnInteger());"
    );
    return 0;
}
```

Console output:

```
result : 10
```

Return Float

To register a C function that returns a Float it must specify *zetscript::zs_float* as return type.

Example

```
#include "zetscript.h"

zetscript::zs_float returnFloat(
    zetscript::ScriptEngine *_script_engine
){
    return 10.5;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers the C function
    script_engine.registerFunction("returnFloat",returnFloat);

    // // Compiles and runs a script
    script_engine.compileAndRun(
        "Console::outln(\"result : \"+returnFloat());"
    );
    return 0;
}
```

Console output:

```
result : 10.500000
```

Return String

To register a C function that returns a String it can be done by specifying the following return types,

- *String*
- *StringScriptObject **

Return String as zetscript::String

To register a C function that returns a String it can specify *zetscript::String* as return type.

Example

```
#include "zetscript.h"

zetscript::String returnString(
    zetscript::ScriptEngine *_script_engine
){
    return "Hello world (String)";
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers the C function
    script_engine.registerFunction("returnString",returnString);

    // // Compiles and runs a script
    script_engine.compileAndRun(
        "Console::outln(\"result : \"+returnString());"
    );
    return 0;
}
```

Console output:

```
result : Hello world (String)
```

Return String as zetscript::StringScriptObject *

To register a C function that returns a String it can specify zetscript::StringScriptObject pointer as return type. The instantiation of *zetscript::StringScriptObject* is done throught ScriptEngine instance.

Example

```
#include "zetscript.h"

// ScriptEngine C++ interface function
zetscript::StringScriptObject *returnString(
    zetscript::ScriptEngine *_script_engine
){

    // instance new StringScriptObject using ScriptEngine instance
    zetscript::StringScriptObject *string=_script_engine->newStringScriptObject();

    // set string value "Hello world (StringScriptObject)"
    string->set("Hello world (StringScriptObject)");

    return string;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers the C function
    script_engine.registerFunction("returnString",returnString);

    // // Compiles and runs a script
    script_engine.compileAndRun(
        "Console::outln(\"result : \"+returnString());"
    );

    return 0;
}
```

Console output:

```
result : Hello world (StringScriptObject)
```

Return Array

To register a C function that returns a Array it must specify *zetscript::ArrayScriptObject* * as return type. The instantiation of *zetscript::ArrayScriptObject* * it's done through *ScriptEngine::newArrayScriptObject* using ScriptEngine instance.

Example

```
#include "zetscript.h"

// Definition of the native function interface returnArray
zetscript::ArrayScriptObject *returnArray(
    zetscript::ScriptEngine *_script_engine
){
    // instance new ArrayScriptObject using ScriptEngine instance
    zetscript::ArrayScriptObject *array=_script_engine->newArrayScriptObject();

    // push first value as integer 10
    array->push<zetscript::zs_int>(10);

    // push second value as float 5.5
    array->push<zetscript::zs_float>(5.5);

    // push third value as boolean true
    array->push<bool>(true);

    // push 4rth reference string "Hello Word"
    array->push<const char *>("Hello World");

    // return object array
    return array;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // bind native function returnArray named as 'returnArray'
    script_engine.registerFunction("returnArray",returnArray);

    // Eval script that calls native function 'returnArray'
    script_engine.compileAndRun(
        "Console::outln(\"result : \"+returnArray());"
    );

    return 0;
}
```

Console output:

```
result : [10,5.500000,true,"Hello World"]
```

Return Object

To register a C function that returns a Object it must specify *zetscript::ObjectScriptObject* * as return type. The instantiation of *zetscript::ObjectScriptObject* * it's done through *ScriptEngine::newObjectScriptObject* using ScriptEngine instance.

Example

```
#include "zetscript.h"

// Definition of the native function interface returnObject
zetscript::ObjectScriptObject *returnObject(
    zetscript::ScriptEngine *_script_engine
){
    // instance new ObjectScriptObject using ScriptEngine instance
    zetscript::ObjectScriptObject *object=_script_engine->newObjectScriptObject();

    // instance new ArrayScriptObject using ScriptEngine instance
    zetscript::ArrayScriptObject *array=_script_engine->newArrayScriptObject();

    // set field "a" as integer 10
    object->set<zetscript::zs_int>("a",10);

    // set field "b" as float 5.5
    object->set<zetscript::zs_float>("b",5.5);

    // set field "c" as boolean true
    object->set<bool>("c",true);

    // set field "d" as string
    object->set<const char *>("d","Hello World");

    // initialize vector object and set as "d"
    for(int i=0; i < 10; i++){
        array->push<zetscript::zs_int>(i);
    }

    object->set<zetscript::ArrayScriptObject *>("d",array);

    // return object
    return object;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // bind native function returnObject named as 'returnObject'
    script_engine.registerFunction("returnObject",returnObject);

    // Eval script that calls native function 'returnObject'
    script_engine.compileAndRun(
        "Console::outln(\"result : \"+returnObject());"
    );

    return 0;
}
```

Console output:

```
result : {"a":10,"b":5.500000,"c":true,"d":[0,1,2,3,4,5,6,7,8,9]}
```

Return registered type

To register a C function that returns a registered type it can be done by specifying the following return types,

- Return registered type by its reference
- Return registered type by ClassScriptObject

Return registered type by its reference

To register a C function that returns a registered type it must specify its native pointer type as return.

> ZetScript **NEVERS** destroy a native pointer returned from registered function. This is why it **CANNOT** allocate and return a type from the C function because this becomes a memory leak. Instead, the object should be instantiated outside of ZetScript and accessible from a location where its lifetime is active during the execution of the script.

Because the function it returns a native reference the type can be registered as INSTANTIABLE or NON INSTANTIABLE

Example

In the following example it registers a type *Number* as NON INSTANTIABLE (This is because in this example it doesn't need to instantiate any new object in the script code) and property *Number::value* (see ScriptEngine::registerMemberPropertyMetamethod). Finally, it registers C function *returnNumber* that returns a *Number* *. In the main program instantiates *Number* object and is acceded globally, so is active during the execution of the script.

```
#include "zetscript.h"

// C++ class to be registered
class Number{
public:
    float value=0;
};

// Defines getter property for Number::value
zetscript::zs_float NumberZs_get_value(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    return _this->value;
}

Number *_number=NULL;

// C function that returns a Number type pointer
Number *returnNumber(
    zetscript::ScriptEngine *_script_engine
){
    // return global _number
    return _number;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Creates number and init its field value
    _number=new Number();
    _number->value=10;

    // Registers class Number as non instantiable type
    script_engine.registerType<Number>("Number");

    // registers property getter Number::value
    script_engine.registerMemberPropertyMetamethod<Number>("value","_get",NumberZs_get_value);

    // registers C function that returns a Number type pointer
    script_engine.registerFunction("returnNumber",returnNumber);

    // Eval script that C function and prints the result by console
    script_engine.compileAndRun(
        "Console::outln(\"result : \"+returnNumber());"
    );

    // Deletes _number
    delete _number;

    return 0;
```

```
}
```

Console output:

```
result : {"value":10.000000}
```

Return registered type by ClassScriptObject *

To register a C function that returns a registered type it must return *zetscript::ClassScriptObject **. To instantiate *zetscript::ClassScriptObject ** is done through *ScriptEngine::newClassObject* using ScriptEngine instance and the registered class instantiated.

A difference from returning a pointer type of a registered type, when the C function returns a *zetscript::ClassScriptObject **, ZetScript is in charge to deallocate automatically when it has no more references to it.

> The action of return a *zetscript::ClassScriptObject ** ZetScript it considers the instance of a new object in the script code so it needs to register the type as INSTANTIABLE.

Example

In the following example it registers type *Number* as INSTANTIABLE and the property *Number::value*. Finally, it registers C function *returnNumber* that instances and returns *ClassScriptObject* that wraps *Number* type.

```
#include "zetscript.h"

// C++ class to be registered
class Number{
public:
    float value;

    Number(){
        value=0;
    }
};

// defines new function for Number object
Number *NumberZs_new(
    zetscript::ScriptEngine *_script_engine
){
    return new  Number();
}

// defines getter property for Number::x
zetscript::zs_float NumberZs_get_value(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    return _this->value;
}

// defines delete function for Number object
void NumberZs_delete(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    delete _this;
}

// C function that returns classObject
zetscript::ClassScriptObject *returnNumber(
    zetscript::ScriptEngine *_script_engine
){
    // Define script class object
    zetscript::ClassScriptObject *class_object=NULL;

    // Instances number
    Number *number=new Number();

    // initializes value
    number->value=10;

    // instance new ClassScriptObject using ScriptEngine instance and number instance
    class_object=_script_engine->newClassScriptObject(number);

    // return class script object
    return class_object;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Register class Number as instanciable
    script_engine.registerType<Number>("Number",NumberZs_new,NumberZs_delete);
```

```
    // register property getter Number::value
    script_engine.registerMemberPropertyMetamethod<Number>("value","_get",NumberZs_get_value);

    // register C function that returns Number ClassScriptObject
    script_engine.registerFunction("returnNumber",returnNumber);

    // Eval script that C function and prints the result by console
    script_engine.compileAndRun(
        "Console::outln(\"result : \"+returnNumber());"
    );

    return 0;
}
```

Console output:

```
result : {"value":10.000000}
```

Return ANY

To register a C function that returns ANY value is done through ScriptEngine::pushStackElement. This method can also allow to return more than one value.

Example

In the following example it returns a Integer,Float and a String.

```
#include "zetscript.h"

void returnAny(zetscript::ScriptEngine *_script_engine){

    // Converts primitive type to stack element
    zetscript::StackElement r1=_script_engine->toStackElement<zetscript::zs_int>(10);
    zetscript::StackElement r2=_script_engine->toStackElement<zetscript::zs_float>(10.5);
    zetscript::StackElement r3=_script_engine->toStackElement<zetscript::StringScriptObject *>(
        new zetscript::StringScriptObject(_script_engine,"Hello World!!!")
    );

    // Push stack stack elements FIFO order (First into stack First out to be readed in ZetScript)
    _script_engine->pushStackElement(r1);
    _script_engine->pushStackElement(r2);
    _script_engine->pushStackElement(r3);
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers the C function
    script_engine.registerFunction("returnAny",returnAny);

    // // Compiles and runs a script
    script_engine.compileAndRun(
        "var r1,r2,r3;\n"
        "r3,r2,r1 = returnAny(); // Get the values from 'returnAny' from right (the first) to left (the last)  \n"
        "Console::outln(\"r1:{0} r2:{1} r3:{2}\",r1,r2,r3);"
    );
    return 0;
}
```

Console output:

```
r1:10 r2:10.500000 r3:Hello World!!!
```

3.2.2. Parameter types

This section will explain how to register C functions that accepts the following types as parameters,

- Boolean
- Integer
- Float
- String
- Array
- Object
- Function
- Registered type
- ANY

Parameter Boolean

To register a C function that accepts a Boolean it must specify *bool** as parameter type.

Example

```
#include "zetscript.h"


void paramBool(
    zetscript::ScriptEngine *_script_engine
    , bool *_bool
){
    printf("Result : %s\n",*_bool?"true":"false");
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Register C function
    script_engine.registerFunction("paramBool",paramBool);

    // Compiles and runs script
    script_engine.compileAndRun(
        "paramBool(true);"
    );


    return 0;
}
```

Console output:

```
Result : true
```

Parameter Integer

To register a C function that accepts a Integer it can specify *zetscript::zs_int* or *zetscript::zs_int* * as parameter type.

Example

```
#include "zetscript.h"

// The C function to register that prints the result of adding (_numbe1) + (*_number2)
void add(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::zs_int _number1
    ,zetscript::zs_int  *_number2
){
    printf("Result _number1 + _number2: %ld\n",(long int)_number1 + *_number2);
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Registers C function
    script_engine.registerFunction("add",add);

    // Evaluates the script that calls C function passing '10' and '20' values as arguments
    script_engine.compileAndRun(
        "add(10,20);"
    );


    return 0;
}
```

Console output:

```
Result _number1 + _number2: 30
```

Parameter Float

To register a C function that accepts a Float it must specify *zs_float* * as parameter type.

Example

```
#include "zetscript.h"

// c function
void funParamFloat(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::zs_float *_number
){
    printf("Result : %f\n",*_number);
}

int main(){
    zetscript::ScriptEngine script_engine;

    script_engine.registerFunction("funParamFloat",funParamFloat);

    // call c function
    script_engine.compileAndRun(
        "funParamFloat(10.5);"
    );


    return 0;
}
```

Console output:

```
Result : 10.500000
```

Parameter String

To register a C function that accepts a String it can specify *const char **, *String ** or *StringScriptObject ** as a parameter type,

Example

```
#include "zetscript.h"

// c function
void paramString(
    zetscript::ScriptEngine *_script_engine, zetscript::String *_string
){
    printf("Result : '%s' (String *)\n",_string->toConstChar());
}

int main(){
    zetscript::ScriptEngine script_engine;

    script_engine.registerFunction("paramString",paramString);

    // call c function
    script_engine.compileAndRun(
        "paramString(\"Hello world!\");"
    );

    return 0;
}
```

Console output:

```
Result : 'Hello world!' (String *)
```

Parameter Array

To register a C function that accepts a Array it must specify *zetscript::ArrayScriptObject* * as parameter type.

Example

```
#include "zetscript.h"

// c function expects an array of integers
void paramArrayScriptObject(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ArrayScriptObject *_array
){
    printf("Values in array => ");
    for(int i=0; i < _array->length(); i++){
        printf(" %i",(int)_array->get<zetscript::zs_int>(i));
    }
    printf("\n");
}

int main(){
    zetscript::ScriptEngine script_engine;

    script_engine.registerFunction("paramArrayScriptObject",paramArrayScriptObject);

    // call c function with string param
    script_engine.compileAndRun(
        "paramArrayScriptObject(["
        "    0,1,2,3,4,5"
        "]);"
    );

    return 0;
}
```

Console output:

```
Values in array =>  0 1 2 3 4 5
```

Parameter Object

To register a C function that accepts a Object it must specify *zetscript::ObjectScriptObject** as parameter type.

Example

```
#include "zetscript.h"

// c function expects an array of integers and floats
void paramObjectScriptObject(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::ObjectScriptObject *_object
){
    printf("Values in object:\n");
    auto keys=_object->getKeys();
    for(int i=0; i < keys.length();i++){
        printf(
            "Key: '%s' => Value: %i\n"
            ,keys.get(i).toConstChar()
            ,(int)_object->get<zetscript::zs_int>(keys.get(i))
        );
    }
}

int main(){
    zetscript::ScriptEngine script_engine;

    script_engine.registerFunction("paramObjectScriptObject",paramObjectScriptObject);

    // Calls registered "paramObjectScriptObject" with an Object as a parameter
    script_engine.compileAndRun(
        "paramObjectScriptObject({"
        "    a:0"
        "     ,b:1"
        "     ,c:2"
        "     ,d:3"
        "     ,e:4"

        "});"
    );


    return 0;
}
```

Console output:

```
Values in object:
Key: 'a' => Value: 0
Key: 'b' => Value: 1
Key: 'c' => Value: 2
Key: 'd' => Value: 3
Key: 'e' => Value: 4
```

Parameter Function

To register a C function that accepts a Function it must specify *zetscript::ScriptFunction* * as parameter type. To make *Function* callable from C++ must use ScriptEngine::bindScriptFunction

Example

The following example it registers function *paramFunction* that accepts a Function as parameter. Later, it evaluates an script that calls *paramFunction* passing anonymous function. In the calling function *paramFunction* it binds the anonymous function and it calls.

```
#include "zetscript.h"

// The C function to register that binds and calls the script function passed by parameter
void paramFunction(
    zetscript::ScriptEngine *_script_engine,
    zetscript::ScriptFunction *_script_function
){

    // bind script function to make it callable
    auto script_function=_script_engine->bindScriptFunction<void ()>(_script_function);

    // call script function
    script_function();
}

int main(){
    zetscript::ScriptEngine script_engine;

    // register C function
    script_engine.registerFunction("paramFunction",paramFunction);

    // Evaluates the script that calls C function with anonymous function as argument
    script_engine.compileAndRun(
        "paramFunction(function(){\n"
        "    Console::out(\"Calling from script function\")\n"
        "})\n"
    );


    return 0;
}
```

Console output:

```
Calling from script function
```

Parameter Registered Type

To register a C function that accepts a registered type as a parameter, it must specify native pointer type as parameter type.

Example

In the following example it registers the *Number* type, property *Number::value* and finally registers function *mul10Number* that accepts a *Number* * type as parameter. In the script execution instances a *Number* that later it is passed as argument to *mul10Number* and multiplies its property *value* by 10.

```
#include "zetscript.h"

// C++ class to be registered
class Number{
public:
    float value;

    Number(){
        value=0;
    }
};

// defines new function for Number object
Number *NumberZs_new(
    zetscript::ScriptEngine *_script_engine
){
    return new  Number();
}

// defines setter property for Number::x
void NumberZs_set_value(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
    , zetscript::zs_float *_value
){
    _this->value=*_value;
}

// defines getter property for Number::x
zetscript::zs_float NumberZs_get_value(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    return _this->value;
}

// defines delete function for Number object
void NumberZs_delete(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    delete _this;
}

// C function the accepts native Number
void mul10Number(
    zetscript::ScriptEngine *_script_engine
    , Number *_number
){
    // initialize x and y
    _number->value*=10;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Register class Number
    script_engine.registerType<Number>("Number",NumberZs_new,NumberZs_delete);

    // Register property setter Number::x
    script_engine.registerMemberPropertyMetamethod<Number>("value","_set",NumberZs_set_value);

    // Register property getter Number::x
    script_engine.registerMemberPropertyMetamethod<Number>("value","_get",NumberZs_get_value);

    // Register native function mulNumber named as 'mulNumber'
    script_engine.registerFunction("mul10Number",mul10Number);

    // Eval script that calls native function 'mulNumber'
```

```
    script_engine.compileAndRun(
        "var number=new Number();\n"
        "number.value=10;\n"
        "Console::outln(\"before : \"+number);\n"
        "mul10Number(number)\n"
        "Console::outln(\"after call 'mul10Number': \"+number);"
    );

    return 0;
}
```

Console output:

```
before : {"value":10.000000}
after call 'mul10Number': {"value":100.000000}
```

Parameter ANY

To register a C function that accepts ANY type of parameter it must specify zetscript::StackElement pointer as parameter type.

Example

This example it calls a registered function paramAny and pass an Integer,Float and a String on the same function.

```
#include "zetscript.h"

void paramAny(
    zetscript::ScriptEngine *_script_engine
    ,zetscript::StackElement *_stack_element
){
    printf("StackElement passed is type '%s' with value of '%s'\n"
        ,_script_engine->stackElementTypeToString(_stack_element).toConstChar()
        ,_script_engine->stackElementValueToString(_stack_element).toConstChar()
    );
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Register C function
    script_engine.registerFunction("paramAny",paramAny);

    // Compiles and runs script
    script_engine.compileAndRun(
        "paramAny(1); // pass Integer value to paramAny\n"
        "paramAny(5.2); // pass Float value to paramAny \n"
        "paramAny(\"Hello world!\"); // pass String value to paramAny \n"
    );


    return 0;
}
```

Console output:

```
StackElement passed is type 'Integer' with value of '1'
StackElement passed is type 'Float' with value of '5.200000'
StackElement passed is type 'String' with value of 'Hello world!'
```

3.3. Call ZetScript from C++

To make a ZetScript function callable from C++ is done through ScriptEngine::bindScriptFunction.

3.3.1. Return types

This section will explain how to bind script functions that return the following types,

- Boolean
- Integer
- Float
- String
- Array
- Object
- Instance of class type object
- Instance of registered type object

Return Boolean

To bind a ZetScript function that returns a Boolean it must specify *bool* return type on its function signature.

Example

```
#include "zetscript.h"

int main()
{
    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'returnBoolean' that returns 'Boolean' value
    script_engine.compile(
        "function returnBoolean(){\n"
        "    return true;\n"
        "}\n"
    );

    // It binds 'returnBoolean' as 'bool(void)'
    auto returnBoolean=script_engine.bindScriptFunction<bool()>("returnBoolean");

    // Calls ZetScript function and prints return value by console.
    printf("result : %s\n",returnBoolean()?"true":"false");

    return 0;
}
```

Console output:

```
result : true
```

Return Integer

To bind a ZetScript function that returns a Integer it must specify *zetscript::zs_int* return type on its function signature.

Example

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'returnInteger' that returns 'Integer' value
    script_engine.compile(
        "function returnInteger(){\n"
        "    return 10;\n"
        "}\n"
    );

   // It binds 'returnInteger' as 'zs_int(void)'
  auto returnInteger=script_engine.bindScriptFunction<
    zetscript::zs_int()
  >("returnInteger");

   // Calls ZetScript function and prints return value by console.
   printf("result : %ld\n",(long int)returnInteger());
    return 0;
}
```

Console output:

```
result : 10
```

Return Float

To bind a ZetScript function that returns a Float it must specify *zetscript::zs_float* return type on its function signature.

Example

```
#include "zetscript.h"

int main()
{
    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'returnFloat' that returns 'Float' value
    script_engine.compile(
        "function returnFloat(){\n"
        "    return 10.5;\n"
        "}\n"
    );

    // It binds 'returnFloat' as 'zs_float(void)'
    auto returnFloat=script_engine.bindScriptFunction<zetscript::zs_float()>("returnFloat");

    // Calls ZetScript function and prints return value by console.
    printf("result : %f\n",returnFloat());

    return 0;
}
```

Console output:

```
result : 10.500000
```

Return String

To bind a ZetScript function that returns a String it can be done by specifying the following return types in its function signature,

- String
- StringScriptObject *

Return String as zetscript::String

To bind a ZetScript function that returns a String it can specify *zetscript::String* return type on its function signature.

Example

```
#include "zetscript.h"

int main(){
  zetscript::ScriptEngine script_engine;

  // Evaluates ZetScript function 'returnString' that returns 'String' value
  script_engine.compile(
        "function returnString(){\n"
        "    return \"Hello World\";\n"
        "}\n"
    );

   // It binds 'returnString' as 'String(void)'
  auto returnString=script_engine.bindScriptFunction<
    zetscript::String()
  >("returnString");

  // Calls ZetScript function and prints return value by console.
  printf("result : %s\n",returnString().toConstChar());
    return 0;
}
```

Console output:

```
result : Hello World
```

Return String as zetscript::StringScriptObject *

To bind a ZetScript function that returns a String it can specify *zetscript::StringScriptObject* * return type on its function signature.

After using the object returned by the function it has to decrease its reference count by calling *ScriptEngine::unrefLifeTimeObject*.

Example

```
#include "zetscript.h"

int main(){

  zetscript::ScriptEngine script_engine;

  // Evaluates ZetScript function 'returnString' that returns 'String' value
  script_engine.compile(
  "function returnString(){\n"
      "    return \"Hello World\";\n"
      "}\n"
    );

  // It binds 'returnString' as 'StringScriptObject *(void)'
  auto returnString=script_engine.bindScriptFunction<
    zetscript::StringScriptObject *()
  >("returnString");

  // Calls ZetScript function which it returns 'StringScriptObject *' reference
  auto object=returnString();

  // Prints its value by console.
  printf("result : %s\n",object->get().toConstChar());

  // 'unrefLifetimeObject' it decreases the reference count of the script object to tell is not used anymore
  script_engine.unrefLifetimeObject(object);

    return 0;
}
```

Console output:

```
result : Hello World
```

Return Array

To bind a ZetScript function that returns a Array it must specify *zetscript::ArrayScriptObject* * return type on its function signature.

After using the object returned by the function it has to decrease its reference count by calling *ScriptEngine::unrefLifeTimeObject*.

Example

```
#include "zetscript.h"

int main()
{
    zetscript::ScriptEngine script_engine;

  // Evaluates ZetScript function 'returnArray' that returns 'Array' value
  script_engine.compile(
  "function returnArray(){\n"
      "    return [1,true,\"String\"];\n"
     "}\n"
  );

  // It binds 'returnArray' as 'ArrayScriptObject *(void)'
  auto returnArray=script_engine.bindScriptFunction<
    zetscript::ArrayScriptObject *()
  >("returnArray");

  // Calls ZetScript function which it returns 'ArrayScriptObject *' reference
  auto array_object=returnArray();

  // Prints its value by console.
  printf("result : %s\n",array_object->toString().toConstChar());

   // 'unrefLifetimeObject' it decreases the reference count of the script object to tell is not used anymore
   script_engine.unrefLifetimeObject(array_object);

    return 0;
}
```

Console output:

```
result : [1,true,"String"]
```

Return Object

To bind a ZetScript function that returns a Object it must specify *zetscript::ObjectScriptObject* * return type on its function signature.

After using the object returned by the function it has to decrease its reference count by calling *ScriptEngine::unrefLifeTimeObject*.

Example

```
#include "zetscript.h"

int main()
{
    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'returnObject' that returns 'ScriptObject' value
    script_engine.compile(
        "function returnObject(){\n"
        "    return {a:1,b:true,c:\"String\"};\n"
        "}\n"
    );

   // It binds 'returnObject' as 'ObjectScriptObject *(void)'
   auto returnObject=script_engine.bindScriptFunction<
      zetscript::ObjectScriptObject *()
   >("returnObject");

   // Calls ZetScript function which it returns 'ObjectScriptObject *' reference
   auto object_object=returnObject();

   // Prints its value by console.
   printf("result : %s\n",object_object->toString().toConstChar());

   // 'unrefLifetimeObject' it decreases the reference count of the script object to tell is not used anymore
   script_engine.unrefLifetimeObject(object_object);

    return 0;
}
```

Console output:

```
result : {"a":1,"b":true,"c":"String"}
```

Return instance of class type object

To bind a ZetScript function that returns an instance of Class type object it can specify *zetscript::ObjectScriptObject* * or *zetscript::ClassScriptObject* * return type on its function signature.

After using the object returned by the function it has to decrease its reference count by calling *ScriptEngine::unrefLifeTimeObject*.

Example

```
#include "zetscript.h"

int main(){

    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript class Number and function 'newNumber' that returns a new instance of type 'Number'
    script_engine.compile(
      "class Number{ \n"
      "   var value=0;\n"
      "}\n"
      "\n"
      "// 'returnNumber' instances 'Number' type\n"
      "\n"
      "function newNumber(){\n"
      "    return new Number();\n"
      "}\n"
    );

    // It binds 'newNumber' as '(ObjectScriptObject *)(void)'
  auto newNumber=script_engine.bindScriptFunction<
    zetscript::ObjectScriptObject *()
  >("newNumber");

   // Calls ZetScript function which it returns 'ObjectScriptObject *' reference
    auto number=newNumber();

    // Prints return value by console.
    printf("result : %s\n",number->toString().toConstChar());

   // 'unrefLifetimeObject' it decreases the reference count of the script object to tell is not used anymore
   script_engine.unrefLifetimeObject(number);

    return 0;
}
```

Console output:

```
result : {"value":0}
```

Return instance of registered type object

To bind a ZetScript function that returns an instance of registered type object it must specify *zetscript::ClassScriptObject* * return type on its function signature. After returning *zetscript::ClassScriptObject* *, it can fetch its C++ registered type by calling *ClassScriptObject::to<_T *>*, where _T is the C++ type.

After using the object returned by the function it has to decrease its reference count by calling *ScriptEngine::unrefLifeTimeObject*.

Example

In this example, it registers the *Number* type and the property *Number::value* (see ScriptEngine::registerMemberPropertyMetamethod). In the example also, it can see the implementation of a function 'newNumber' in the ZetScript code that returns a new *Number*. After evaluate the script code, it binds the script function *newNumber* into a in C++ variable called *newNumber*. Finally, it calls *newNumber* from C++ and save the *ClassScriptObject* type returned into C++ *class_object_number* variable to print its content information.

```
#include "zetscript.h"

// Class Number to register
class Number{
public:
    float value;
    Number(){
        value=0;
    }
};

// defines new function Number ClassScriptObject
Number *NumberZs_new(
    zetscript::ScriptEngine *_script_engine
){
    return new  Number();
}

void NumberZs_constructor(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
    , zetscript::zs_int _value
){
    _this->value=_value;
}

// defines getter property Number::x ClassScriptObject
zetscript::zs_int NumberZs_get_value(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    return _this->value;
}

// defines delete function Number ClassScriptObject
void NumberZs_delete(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    delete _this;
}

int main(){
    zetscript::ScriptEngine script_engine;

    // Register class 'Number' as instantiable
    script_engine.registerType<Number>("Number",NumberZs_new,NumberZs_delete);

    // Register constructor
    script_engine.registerConstructor<Number>(NumberZs_constructor);

    // register property getter Number::value
    script_engine.registerMemberPropertyMetamethod<Number>("value","_get",NumberZs_get_value);

    // Evaluates function 'newNumber' that returns an instance of registered type 'Number'
    script_engine.compile(
        "// 'newNumber' instances a new 'Number' type\n"
        "function newNumber(){\n"
        "   return new Number(10);\n"
        "}\n"
    );
```

```
    // It binds 'newNumber' as '(ClassScriptObject *)(void)'
    auto newNumber=script_engine.bindScriptFunction<
        zetscript::ClassScriptObject *()
    >("newNumber");

    // Calls ZetScript function which it returns 'ClassScriptObject *' reference
    auto class_object_number=newNumber();

    // Prints the contents by console.
    printf("From zetscript object : %s\n",class_object_number->toString().toConstChar());

    // Cast C++ 'Number' type pointer
    auto number=class_object_number->to<Number *>();

    // Prints Number's properties by console.
    printf("From C++ pointer type : number->value=%f\n",number->value);

    // 'unrefLifetimeObject' it decreases the reference count of script object to tell is not used anymore
    script_engine.unrefLifetimeObject(class_object_number);
    return 0;
}
```

Console output:

```
From zetscript object : {"value":10}
From C++ pointer type : number->value=10.000000
```

3.3.2. Parameter types

This section will explain how to bind script functions that accepts the following types as parameters,

- Boolean
- Integer
- Float
- String
- Array
- Object
- Function
- Registered class

Parameter Boolean

To bind a ZetScript function that accepts a Boolean it must specify *bool * as parameter type on its function signature.

Example

```
#include "zetscript.h"

int main()
{
    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'paramBoolean' that prints the contents of '_value'
    script_engine.compile(
        "function paramBoolean(_value){\n"
        "    Console::outln(\"result : \"+_value);\n"
        "}\n"
    );

    // It binds 'paramBoolean' as '(void *)(bool *)'
    auto paramBoolean=script_engine.bindScriptFunction<void(bool *)>("paramBoolean");

    // Prepare parameter values
    bool value=true;

    // Calls binded ZetScript function with parameters
    paramBoolean(&value);

    return 0;
}
```

Console output:

```
result : true
```

Parameter Integer

To bind a ZetScript function that accepts a Integer it must specify *zetscript::zs_int* or *zetscript::zs_int* * as parameter type on its function signature.

Example

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'add' that prints the result of the operation '+' from two arguments
    script_engine.compile(
        "function printAdd(_value1,_value2){\n"
        "    Console::outln(\"result : \"+(_value1 + _value2));\n"
        "}\n"
    );

   // It binds 'printAdd' as '(void *)(zs_int, zs_int)' (by value)
   auto printAddByValue=script_engine.bindScriptFunction<
      void(
         zetscript::zs_int _value1
         ,zetscript::zs_int _value2
      )
   >("printAdd");

    // Calls binded ZetScript function with parameters by value
    printAddByValue(10,10);

    return 0;
}
```

Console output:

```
result : 20
```

Parameter Float

To bind a ZetScript function that accepts a Float it must specify or *zetscript::zs_float** as parameter type on its function signature.

Example

```
#include "zetscript.h"

int main(){
   zetscript::ScriptEngine script_engine;

   // Evaluates ZetScript function 'printAdd' that prints the result of the operation '+' from two arguments
   script_engine.compile(
        "function printAdd(_value1,_value2){\n"
        "    Console::outln(\"result : \"+(_value1 + _value2));\n"
        "}\n"
    );

  // It binds 'printAdd' as '(void *)(zs_float, zs_float *)'
  auto printAdd=script_engine.bindScriptFunction<
    void(
      zetscript::zs_float *_value1
      ,zetscript::zs_float *_value2
    )>("printAdd");

  // Prepare parameter values
  zetscript::zs_float value1=3.5;
  zetscript::zs_float value2=10.7;

  // Calls binded ZetScript function with parameters
   printAdd(&value1,&value2);

    return 0;
}
```

Console output:

```
result : 14.200000
```

Parameter String

To bind a ZetScript function that accepts a String it can specify the following types as a parameter on its function signature,

- const char *
- zetscript::String *
- zetscript::StringScriptObject *

Parameter String as const char *

To bind a ZetScript function that accepts a String it can specify *const char* * as parameter type on its function signature.

Example

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine;

   // Evaluates ZetScript function 'concat' that prints the result of the operation '+' from two arguments
   script_engine.compile(
        "function printConcat(_value1, _value2){\n"
        "    Console::outln(\"result : \"+_value1+\" \"+_value2);\n"
        "}\n"
    );
   // It binds 'concat' as '(void *)(const char *, const char *)'
   auto printConcat=script_engine.bindScriptFunction<void(const char * _value1, const char * _value2)>("printConcat");

   // Calls binded ZetScript function with parameters
   printConcat("Hello","World");
    return 0;
}
```

Console output:

```
result : Hello World
```

Parameter String as zetscript::String *

To bind a ZetScript function that accepts a String it can specify *zetscript::String* * as parameter type on its function signature.

Example

```
#include "zetscript.h"

int main(){
    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'concat' that prints the result of the operation '+' from two arguments
    script_engine.compileAndRun(
        "function printConcat(_value1, _value2){\n"
        "    Console::outln(\"result : \"+_value1+\" \"+_value2);\n"
        "}\n"
    );

  // It binds 'printConcat' as '(void *)(String *, String *)'
  auto printConcat=script_engine.bindScriptFunction<
    void(
    zetscript::String * _value1, zetscript::String * _value2
  )>("printConcat");

  // Prepare param values
  zetscript::String value1="Hello";
  zetscript::String value2="World";

  // Calls ZetScript function by value
  printConcat(&value1,&value2);
    return 0;
}
```

Console output:

```
result : Hello World
```

Parameter String as zetscript::StringScriptObject *

To bind a ZetScript function that accepts a String it can specify *zetscript::StringScriptObject* * as parameter type on its function signature.

Example

```
#include "zetscript.h"

int main(){

    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'printConcat' that prints the result of the operation '+' from two arguments
    script_engine.compile(
        "function printConcat(_value1, _value2){\n"
        "    Console::outln(\"result : \"+_value1+\" \"+_value2);\n"
        "}\n"
    );

    // It binds 'printConcat' as '(void *)(StringScriptObject *, StringScriptObject *)'
   auto printConcat=script_engine.bindScriptFunction<
      void(
         zetscript::StringScriptObject * _value1
         ,zetscript::StringScriptObject * _value2
      )
   >("printConcat");

    // Prepare param values
    auto value1=script_engine.newStringScriptObject("Hello");
    auto value2=script_engine.newStringScriptObject("World");

    // Calls binded ZetScript function with parameters
    printConcat(value1,value2);


    return 0;
}
```

Console output:

```
result : Hello World
```

Parameter Array

To bind a ZetScript function that accepts a Array it must specify or *zetscript::ArrayScriptObject* * as parameter type on its function signature.

Example

```
#include "zetscript.h"

int main()
{
    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'printConcat' that prints the result of the operation '+' from two arguments
    script_engine.compile(
        "function printConcat(_value1, _value2){\n"
        "    Console::outln(\"result : \"+(_value1+_value2));\n"
        "}\n"
    );

    // It binds 'printConcat' as '(void *)(ArrayScriptObject *, ArrayScriptObject *)'
    auto printConcat=script_engine.bindScriptFunction<
      void(
         zetscript::ArrayScriptObject * _value1
         ,zetscript::ArrayScriptObject * _value2
      )>("printConcat");

    // Prepare param values
    auto array1=script_engine.newArrayScriptObject();
    auto array2=script_engine.newArrayScriptObject();

   // push values for array1
   array1->push<bool>(false);
   array1->push<zetscript::zs_int>(10);
   array1->push<const char *>("Hello");

   // push values for array2
   array2->push<bool>(true);
   array2->push<zetscript::zs_float>(20.5);
   array2->push<const char *>("World");


    // Calls binded ZetScript function with parameters
    printConcat(array1,array2);


    return 0;
}
```

Console output:

```
result : [false,10,"Hello",true,20.500000,"World"]
```

Parameter Object

To bind a ZetScript function that accepts a Object it must specify *zetscript::ObjectScriptObject* * as parameter type on its function signature.

Example

```
#include "zetscript.h"

int main()
{
    zetscript::ScriptEngine script_engine;

    // Evaluates ZetScript function 'printConcat' that prints the result of the operation '+' from two arguments
    script_engine.compile(
        "function printConcat(_value1, _value2){\n"
        "    Console::outln(\"result : \"+(_value1+_value2));\n"
        "}\n"
    );

    // It binds 'printConcat' as '(void *)(ArrayScriptObject *, ArrayScriptObject *)'
    auto printConcat=script_engine.bindScriptFunction<
      void(
         zetscript::ObjectScriptObject * _value1
         ,zetscript::ObjectScriptObject * _value2
   )>("printConcat");

    // Prepare param values
    auto object1=script_engine.newObjectScriptObject();
    auto object2=script_engine.newObjectScriptObject();

    // push values for object1
    object1->set<bool>("a",true);
    object1->set<zetscript::zs_int>("b",10);
    object1->set<const char *>("c","Hello");

    // push values for object2
    object2->set<bool>("d",false);
    object2->set<zetscript::zs_float>("e",20.5);
    object2->set<const char *>("f","World");


    // Calls binded ZetScript function with parameters
    printConcat(object1,object2);


    return 0;
}
```

Console output:

```
result : {"a":true,"b":10,"c":"Hello","d":false,"e":20.500000,"f":"World"}
```

Parameter registered type

To bind a ZetScript function that accepts a registered type as a parameter, it must specify native pointer type as parameter type on its function signature.

Example

```
#include "zetscript.h"

// C++ class to be registered
class Number{
public:
    float value;

    Number(){
        value=0;
    }

    Number(float _value){
        value=_value;
    }
};

//------------------------------
// REGISTER FUNCTIONS

// defines getter property for Number::value
zetscript::zs_float NumberZs_get_value(
    zetscript::ScriptEngine *_script_engine
    , Number *_this
){
    return _this->value;
}

// REGISTER FUNCTIONS
//------------------------------

int main()
{
    zetscript::ScriptEngine script_engine;

   // Register class Number
    script_engine.registerType<Number>("Number");

    // register property getter Number::x
    script_engine.registerMemberPropertyMetamethod<Number>("value","_get",NumberZs_get_value);


    // Evaluates ZetScript function 'paramNumber' that prints the contents of '_number'
    script_engine.compile(
        "function paramNumber(_number){\n"
        "    Console::outln(\"result : \"+_number);\n"
        "}\n"
    );

    // It binds 'concat' as '(void *)(ArrayScriptObject *, ArrayScriptObject *)'
    auto paramNumber=script_engine.bindScriptFunction<void(Number * _number)>("paramNumber");

    // Prepare parameters
    auto number=Number(10);

    // Calls binded ZetScript function with parameters
    paramNumber(&number);


    return 0;
}
```

Console output:

```
result : {"value":10.000000}
```

3.4. Exposing C++ types to ZetScript

In this chapter will see how to to register C/C++ types, function members, metamethods and properties in order to be exposed in ZetScript through examples.

Most of the examples show in this chapter will be related about the following class *Number*,

```
class Number{
public:

    float value;

    Number(){
        value=0;
    }

    Number(float _value){
        value=_value;
    }

};
```

On each example will mention *NumberZs_register*,

```
void NumberZs_register(ScriptEngine *_script_engine){

    // ...

}
```

This method is the place where it will add the registering functions of *Number*.

3.4.1. Register a type

As it can be explained in section ScriptEngine::registerType it can register as INSTANTIABLE type or as NON INSTANTIABLE type.

This example aims to register *Number* as INSTANTIABLE type, so let's define *NumberZs_new* and *NumberZs_delete* functions that creates and destroys a *Number* type instantiation respectively,

```
Number *NumberZs_new(ScriptEngine *_script_engine){
    return new Number();
}

void NumberZs_delete(ScriptEngine *_script_engine,Number *_this){
    delete _this;
}
```

Then, in the *NumberZs_register* function, it registers *Number* type as instantiable by passing *NumberZs_new* and *NumberZs_delete* functions,

```
void NumberZs_register(ScriptEngine *_script_engine){

    _script_engine->registerType<Number>("Number",NumberZs_new,NumberZs_delete);

}
```

Finally, the following code shows an example of instantiation of a *Number* type,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun("var number=new Number();");

    return 0;
}
```

Register a constructor

To register a constructor function it has to create and register a C function as it explains on ScriptEngine::registerConstructor.

Example

The following code defines two functions that accepts a Float value or *Number* pointer type respectively,

```
//...

void NumberZs_constructor(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value=*_value;
}

void NumberZs_constructor(ScriptEngine *_script_engine,Number *_this,Number *_value){
    _this->value=_value->value;
}
```

Next, the functions *NumberZs_constructor* are registered with *ScriptEngine::registerConstructor* method,

```
void NumberZs_register(ScriptEngine *_script_engine){

    //...

    _script_engine->registerConstructor<Number>(static_cast<void (*)(ScriptEngine *_script_engine,Number *_this, zs_float *_value)>(
&NumberZs_constructor));

    _script_engine->registerConstructor<Number>(static_cast<void (*)(ScriptEngine *_script_engine,Number *_this, Number *_value)>(
&NumberZs_constructor));

    //...

}
```

Finally, the following code it shows an example of a script that creates an object *Number* type passing a float value on its constructor,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun("var number=new Number(10.5);");

    return 0;
}
```

3.4.2. Register members

Register member function

To register a member function it has to create and register a C function explained in section ScriptEngine::registerMemberFunction.

Example

The following code defines a function that returns a Integer,

```
zs_int NumberZs_toInteger(ScriptEngine *_script_engine,Number *_this){
    return _this->value;
}
```

Next, it registers *NumberZs_toInteger* as member function named *toInteger* through with *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("toInteger",&NumberZs_toInteger);

    //...

}
```

Finally, the following code it shows an example of a script that calls member function *Number::toInteger()* after create object *Number* type,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(10.5);\n"
        "Console::outln(\"number.toInteger() : \"+number.toInteger())\n"
    );

    return 0;
}
```

The ouput is the following,

```
number.toInteger() : 10
```

Register static member function

To register a static member function it has to create and register a C function explained in section ScriptEngine::registerStaticMemberFunction.

Example

The following code defines a function *NumberZs_pow* that takes two arguments and, returns the power raised to the base number,

```
//...

zs_float NumberZs_pow(ScriptEngine *_script_engine,zs_float *_base, zs_float * _power){
    return (*_base) * (*_power);
}
```

Next, it registers *NumberZs_pow* as static member function named *pow* through with *ScriptEngine::registerStaticMemberFunction*,

```
void NumberZs_register(ScriptEngine *_script_engine){

    //...

    _script_engine->registerStaticMemberFunction<Number>("pow",&NumberZs_pow);

    //...

}
```

Finally, the following code it shows an example of a script that calls static member function *Number::pow()*,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"Number::pow(2,10) : \"+Number::pow(2,10))"
    );

    return 0;
}
```

The ouput is the following,

```
Number::pow(2,10) : 20.000000
```

3.4.3. Inheritance

To register a type and tell that is base of other type is done through ScriptEngine::extends.

Example

In this example it shows an example the the registered type *MyCppTypeExtend* is extended from registered type *MyCppType*,

```
#include "zetscript.h"

class MyCppType{
    public:
    int data1;
    int function1(int arg){
        printf("MyCppType::function1 - Argument is %i\n",this->data1+arg);
        return this->data1 + arg;
    }
};

class MyCppTypeExtend:public MyCppType{
public:
    float data2;
    int function2(float arg){
        printf("MyCppTypeExtend::function2 - Float argument is %.02f\n",this->data2 + arg);
        return this->data2 + arg;
    }
};

zetscript::zs_int MyCppType_function1(
    zetscript::ScriptEngine *_script_engine
    , MyCppType *_this
    ,zetscript::zs_int _arg
){
    return _this->function1(_arg);
}

MyCppTypeExtend * MyCppTypeExtend_new(
    zetscript::ScriptEngine *_script_engine
){
    return new MyCppTypeExtend();
}

void MyCppTypeExtend_constructor(
    zetscript::ScriptEngine *_script_engine
    ,MyCppTypeExtend *_this
    ,zetscript::zs_int _data1){
    _this->data1=_data1;
    _this->data2=_data1*10;
}

zetscript::zs_int MyCppTypeExtend_function2(
    zetscript::ScriptEngine *_script_engine
    ,MyCppTypeExtend *_this
    ,zetscript::zs_int _arg){
    return _this->function1(_arg);
}

void MyCppTypeExtend_delete(
    zetscript::ScriptEngine *_script_engine
    ,MyCppTypeExtend *_this
){
    delete _this;
}

int main(){
    zetscript::ScriptEngine script_engine; // instance zetscript

    // Register MyCppType as MyCppType in script side
    script_engine.registerType<MyCppType>("MyCppType");

    // Register MyCppType::function1
    script_engine.registerMemberFunction<MyCppType>("function1",MyCppType_function1);

    // Register MyCppTypeExtend as MyCppTypeExtend in script side as instantiable
    script_engine.registerType<MyCppTypeExtend>("MyCppTypeExtend",MyCppTypeExtend_new,MyCppTypeExtend_delete);

    // Tells MyCppTypeExtends extends from MyCppType
    script_engine.extends< MyCppTypeExtend,MyCppType >();
```

```
    // Register MyCppTypeExtend::function2
    script_engine.registerMemberFunction<MyCppTypeExtend>("function2",MyCppTypeExtend_function2);

    // Instance object as ScriptMyCppTypeExtend calls object.function1
    script_engine.compileAndRun(
        "class ScriptMyCppTypeExtend extends MyCppTypeExtend{\n"
        "   function1(_arg1){\n"
        "       Console::outln(\"script argument : {0} \",_arg1);\n"
        "       super(_arg1); // calls cpp MyCppType::function1\n"
        "   }\n"
        "};\n"
        "var object=new ScriptMyCppTypeExtend(10);\n"
        "object.function1(5);\n"
    );
    return 0;
}
```

Console output,

```
script argument : 5
MyCppType::function1 - Argument is 5
```

A ZetScript script class can be extended from any registered type. In the following evaluates an script that a script class called *ScriptMyCppTypeExtend* extends from registered type *MyCppTypeExtend*,

```
int main(){

    // ...

    script_engine.compileAndRun(
        "class ScriptMyCppTypeExtend extends MyCppTypeExtend{\n"
        "   function1(_arg1){\n"
        "       Console::outln("script argument is "+_arg1);\n"
        "       super(this.data1+arg1); // calls function1\n"
        "   }\n"
        "};\n"
        "var MyCppType=new ScriptMyCppTypeExtend(10);\n"
        "MyCppType.function1(5);\n"
    );

    return 0;
}
```

Console output,

```
script argument : 5
MyCppType::function1 - Argument is 5
```

3.4.4. Register metamethods

ZetScript can register static and member functions in order to implement metamethods seen in Language Class Metamethods.

Member metamethods

_addassign()

Implements *addition assignment* operator (aka +=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_addassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance.
- *_this* : The current instance.
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs an addition and assignment to the current instance from a *Float* value or *Number* object respectively,

```
void NumberZs_addassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value+=*_n;
}

void NumberZs_addassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value+=_n->value;
}
```

Next, each function *NumberZs_addassign* is registered as member metamethod *_addassign* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerMemberFunction<Number>("_addassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_addassign));

    _script_engine->registerMemberFunction<Number>("_addassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_addassign));
    //...

}
```

Finally, the following code it shows an example of a script that add and assigns values from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number+=20 => {0}\",number+=20)\n"
        "Console::outln(\"number+=new Number(30) => {0}\",number+=new Number(30))\n"
    );

    return 0;
}
```

Console output:

```
number+=20 => 40.00
number+=new Number(30) => 70.00
```

_andassign()

Implements *bitwise AND assignment* operator (aka &=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_andassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs a bitwise AND and assignment to the current instance from a *Float* value or *Number* type respectively,

```
void NumberZs_andassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value-=*_n;
}

void NumberZs_andassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value-=_n->value;
}
```

Next, each function *NumberZs_andassign* is registered as member metamethod *andassign_* through ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerMemberFunction<Number>("_andassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_andassign));

    _script_engine->registerMemberFunction<Number>("_andassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_andassign));
    //...

}
```

Finally, the following code it shows an example of a script that performs a bitwise AND and assignment from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(0x7);\n"
        "Console::outln(\"number&=0x3 => {0}\",number&=0x3);\n"
        "Console::outln(\"number&=new Number(0x1) => {0}\",number&=new Number(0x1));\n"
    );

    return 0;
}
```

Console output:

```
number&=0x3 => 3.00
number&=new Number(0x1) => 1.00
```

_divassign()

Implements *division assignment* operator (aka /=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_divassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs a division and assignment to the current instance from a *Float* value or *Number* type respectively,

```
void NumberZs_divassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value-=*_n;
}

void NumberZs_divassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value-=_n->value;
}
```

Next, each function *NumberZs_divassign* is registered as member metamethod *divassign_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerMemberFunction<Number>("_divassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_divassign));

    _script_engine->registerMemberFunction<Number>("_divassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_divassign));
    //...
}
```

Finally, the following code it shows an example of a script that divides and assigns values from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number/=20 => {0}\",number/=20)\n"
        "Console::outln(\"number/=new Number(30) => {0}\",number/=new Number(30))\n"
    );
    return 0;
}
```

Console output:

```
number/=20 => 1.00
number/=new Number(30) => 0.03
```

_in()

Implements *in* operator

Syntax

```
bool RegisteredType_in(zetscript::ScriptEngine *_script_engine, Data *_this, zs_float *_value);
```

Parameters

- *_value* : Value or variable as value to check whether exist or not in the containing class

Returns

Boolean telling whether the *_value* exist in or not.

Example

Let's define type Data as,

```
class Data{
public:
    std::vector<float> data;
    Data(){
        this->data={0,1,1,10,3,4,6};
    }
};
```

The following code defines a function as a *in* operation by searching a value in the vector of current instance of *Data* type,

```
bool DataZs_in(zetscript::ScriptEngine *_script_engine, Data *_this, zs_float *_value){
    for(auto d : _this->data){
        if(d == *_value){
            return true;;
        }
    }
    return false;
}
```

The following code shows an example of registering functions and a script that performs a *in* operation of a *Data* object.

```
#include "NumberZs.h"

class Data{
public:
    zetscript::Vector<float> data;
    Data(){
        float n[]={0,1,1,10,3,4,6};
        for(size_t i=0; i < ZS_ARRAY_SIZE(n); i++){
            data.push(n[i]);
        }
    }
};

Data *data=NULL;

Data * getData(
    zetscript::ScriptEngine *_script_engine
){
    return data;
}

bool DataZs_in(
    zetscript::ScriptEngine *_script_engine
    , Data *_this
    , zetscript::zs_float *_value
){
    for(int i=0; i < _this->data.length(); i++){
        if(*_value == _this->data.get(i)){
            return true;
        }
    }
    return false;
}
```

```
int main(){

    zetscript::ScriptEngine script_engine;

    data=new Data();

    script_engine.registerType<Data>("Data");
    script_engine.registerFunction("getData",getData);
    script_engine.registerMemberFunction<Data>("_in",DataZs_in);

    script_engine.compileAndRun(
        "var data=getData();\n"
        "if(10 in data){\n"
        "   Console::outln(\"10 is content in data\")\n"
        "}\n"
    );


    delete data;

    return 0;
}
```

Console output:

```
10 is content in data
```

_modassign()

Implements *modulus assignment* operator (aka %=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_modassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs a remainder division and assignment to the current instance from a *Float* value or *Number* type respectively,

```
void NumberZs_modassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value-=*_n;
}

void NumberZs_modassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value-=_n->value;
}
```

Next, each function *NumberZs_modassign* is registered as member metamethod *modassign_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerMemberFunction<Number>("_modassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_modassign));

    _script_engine->registerMemberFunction<Number>("_modassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_modassign));
    //...

}
```

Finally, the following code it shows an example of a script that produces the remainder division and assigns values from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(250);\n"
        "Console::outln(\"number%=30 => {0}\",number%=30)\n"
        "Console::outln(\"number%=new Number(100) => {0}\",number%=new Number(100))\n"
    );

    return 0;
}
```

Console output:

```
number%=30 => 10.00
number%=new Number(100) => 10.00
```

_mulassign()

Implements *multiplication assignment* operator (aka *=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_mulassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs a multiplication and assignment to the current instance from a *Float* value or *Number* type respectively,

```
void NumberZs_mulassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value-=*_n;
}

void NumberZs_mulassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value-=_n->value;
}
```

Next, each function *NumberZs_mulassign* is registered as member metamethod *mulassign_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerMemberFunction<Number>("_mulassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_mulassign));

    _script_engine->registerMemberFunction<Number>("_mulassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_mulassign));
    //...
}
```

Finally, the following code it shows an example of a script that multiples and assigns values from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(2);\n"
        "Console::outln(\"number*=2 => {0}\",number*=2)\n"
        "Console::outln(\"number*=new Number(2) => {0}\",number*=new Number(2))\n"
    );
    return 0;
}
```

Console output:

```
number*=2 => 4.00
number*=new Number(2) => 8.00
```

_neg()

Implements *negate* pre operator (aka -a)

Syntax

```
ClassScriptObject *RegisteredType_neg(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

An object with its negated value

Example

The following code defines a function as a *neg* operation by returning a new *Number* instance with the negate value of current instance,

```
Number * NumberZs_neg(ScriptEngine *_script_engine,Number *_this){
    return new Number(-_this->value);
}
```

Next, the function *NumberZs_neg* is registered as member metamethod *neg_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_neg",&NumberZs_neg);

    //...

}
```

Finally, the following code it shows an example of a script that returns the negate of a *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20)\n"
        "Console::outln(\"-number => \"+ (-number))\n"
    );

    return 0;
}
```

Console output:

```
-number => -20.00
```

_not()

Implements *not* pre operator (aka !)

Syntax

```
bool RegisteredType_not(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

A boolean value as a result of not operation

Example

The following code defines a function as a *not* operation as true when current instance has a value of 0,

```
bool NumberZs_not(ScriptEngine *_script_engine,Number *_this){
    return _this->value == 0;
}
```

Next, the function *NumberZs_not* is registered as member metamethod *not_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_not",&NumberZs_not);

    //...

}
```

Finally, the following code it shows an example of a script that evaluates not condition of a *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number()\n"
        "if(!number){\n"
        "   Console::outln(\"Number is empty\") \n"
        "}\n"
    );

    return 0;
}
```

Console output:

```
Number is empty
```

_orassign()

Implements *bitwise OR assignment* operator (aka |=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_orassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs a bitwise OR and assignment to the current instance from a *Float* value or *Number* type respectively,

```
void NumberZs_orassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value-=*_n;
}

void NumberZs_orassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value-=_n->value;
}
```

Next, each function *NumberZs_orassign* is registered as member metamethod *orassign_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerMemberFunction<Number>("_orassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_orassign));

    _script_engine->registerMemberFunction<Number>("_orassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_orassign));
    //...

}
```

Finally, the following code it shows an example of a script that performs a bitwise OR and assignment from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(0x1);\n"
        "Console::outln(\"number|=0x2 => {0}\",number|=0x2)\n"
        "Console::outln(\"number|=new Number(0x4) => {0}\",number|=new Number(0x4))\n"
    );

    return 0;
}
```

Console output:

```
number|=0x2 => 3.00
number|=new Number(0x4) => 7.00
```

_postdec()

Implements *post decrement* operator (aka a++)

Syntax

```
ClassScriptObject * RegisteredType_postdec(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

Return new type wrapped in *zetscript::ClassScriptObject* with the value before perform post decrement operation

Example

The following code defines a function as a *postdec* operation that decrements the value of current instance and returns the object with the value before *postdec* operation,

```
ClassScriptObject * NumberZs_postdec(ScriptEngine *_script_engine,Number *_this){
    return _script_engine->newClassObject(new Number(_this->value--));
}
```

Next, the function *NumberZs_postdec* is registered as member metamethod *postdec_* *through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_postdec",NumberZs_postdec);

    //...

}
```

Finally, the following code it shows an example of a script that performs a post decrement operation of a *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number-- => {0}\",number--)\n"
    );

    return 0;
}
```

Console output:

```
number-- => 20.00
```

_postinc()

Implements *post increment* operator (aka a++)

Syntax

```
ClassScriptObject * RegisteredType_postinc(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

Return new type wrapped in *zetscript::ClassScriptObject* with the value before perform post increment operation

Example

The following code defines a function as a *postinc* operation that increments the value of current instance and returns the object with the value before *postinc* operation,

```
ClassScriptObject *  NumberZs_postinc(ScriptEngine *_script_engine,Number *_this){
    return _script_engine->newClassObject(new Number(_this->value++));
}
```

Next, the function *NumberZs_postinc* is registered as member metamethod *_postinc* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_postinc",NumberZs_postinc);

    //...

}
```

Finally, the following code it shows an example of a script that performs a post increment operation of a *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number++ => {0}\",number++)\n"
    );

    return 0;
}
```

Console output:

```
number++ => 20.00
```

_predec()

Implements *pre decrement* operator (aka --a)

Syntax

```
ClassScriptObject * RegisteredType_predec(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

Return new type wrapped in *zetscript::ClassScriptObject* with the value after perform post increment operation

Example

The following code defines a function as a *predec* operation that pre decrements the value of current instance and returns the object with the value before *predec* operation,

```
ClassScriptObject  *NumberZs_predec(ScriptEngine *_script_engine,Number *_this){
    return _script_engine->newClassObject(new Number(--_this->value));
}
```

Next, the function *NumberZs_predec* is registered as member metamethod *_predec* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_predec",NumberZs_predec);

    //...

}
```

Finally, the following code it shows an example of a script that performs a pre decrement operation of a *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"--number => {0}\",--number)\n"
    );

    return 0;
}
```

Console output:

```
--number => 19.00
```

_preinc()

Implements *pre increment* operator (aka ++a)

Syntax

```
ClassScriptObject *RegisteredType_preinc(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

Return new type wrapped in *zetscript::ClassScriptObject* with the value after perform post increment operation

Example

The following code defines a function as a *preinc* operation that pre increments the value of current instance and returns the object with the value before *preinc* operation,

```
ClassScriptObject * NumberZs_preinc(ScriptEngine *_script_engine,Number *_this){
    return _script_engine->newClassObject(new Number(++_this->value));
}
```

Next, the function *NumberZs_preinc* is registered as member metamethod *preinc_* through ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_preinc",NumberZs_preinc);

    //...

}
```

Finally, the following code it shows an example of a script that performs a pre increment operation of a *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"++number => {0}\",++number)\n"
    );

    return 0;
}
```

Console output:

```
++number => 21.00
```

_set()

Implements *assignment* operator (aka =) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_set(ScriptEngine * _script_engine, RegisteredType *_this, ParamType * _value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : A value to be set. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions performs an assignment from Float value or *Number* object respectively,

```
void NumberZs_set(ScriptEngine *_script_engine,Number *_this, zs_float *_n){
    _this->value=*_n;
}

void NumberZs_set(ScriptEngine *_script_engine,Number *_this,Number *_n){
     _this->value = _n->value;
}
```

Next, each function *NumberZs_set* function is registered as member metamethod *_set* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_set",static_cast<void (*)(ScriptEngine *_script_engine,Number *, zs_float *)>(
&NumberZs_set));

    _script_engine->registerMemberFunction<Number>("_set",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number *)>(
&NumberZs_set));

    //...

}
```

Finally, the following code it shows an example of a script that assigns values from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var n=new Number(0);\n"
        "Console::outln(\"n=10 => {0}\",n=10)\n"
        "Console::outln(\"n=new Number(20) => {0}\",n=new Number(20))\n"
    );

    return 0;
}
```

Console output:

```
n=10 => 10
n=new Number(20) => 20.00
```

_shlassign()

Implements *bitwise shift left assignment* operator (aka <<=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_shlassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs a bitwise shift right and assignment to the current instance from a *Float* value or *Number* type respectively,

```
void NumberZs_shlassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value-=*_n;
}

void NumberZs_shlassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value-=_n->value;
}
```

Next, each function *NumberZs_shlassign* is registered as member metamethod *shlassign_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerMemberFunction<Number>("_shlassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_shlassign));

    _script_engine->registerMemberFunction<Number>("_shlassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_shlassign));
    //...

}
```

Finally, the following code it shows an example of a script that perform bitwise shift right and assignment from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(0x1);\n"
        "Console::outln(\"number<<=1 => {0}\",number<<=1)\n"
        "Console::outln(\"number<<=new Number(1) => {0} \",number<<=new Number(1))\n"
    );

    return 0;
}
```

Console output:

```
number<<=1 => 2.00
number<<=new Number(1) => 4.00
```

_shrassign()

Implements *bitwise shift right assignment* operator (aka >>=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_shrassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs a bitwise shift right and assignment to the current instance from a *Float* value or *Number* type respectively,

```
void NumberZs_shrassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value-=*_n;
}

void NumberZs_shrassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value-=_n->value;
}
```

Next, each function *NumberZs_shrassign* is registered as member metamethod *_shrassign* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerMemberFunction<Number>("_shrassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_shrassign));

    _script_engine->registerMemberFunction<Number>("_shrassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_shrassign));
    //...

}
```

Finally, the following code it shows an example of a script that perform bitwise shift right and assignment from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(0x10);\n"
        "Console::outln(\"number>>=1 => {0}\",number>>=1)\n"
        "Console::outln(\"number>>=new Number(1) => {0}\",number>>=new Number(1))\n"
    );

    return 0;
}
```

Console output:

```
number>>=1 => 8.00
number>>=new Number(1) => 4.00
```

_subassign()

Implements *substraction assignment* operator (aka -=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_subassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs a substraction and assignment to the current instance from a *Float* value or *Number* type respectively,

```
void NumberZs_subassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value-=*_n;
}

void NumberZs_subassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value-=_n->value;
}
```

Next, each function *NumberZs_subassign* is registered as member metamethod *subassign_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerMemberFunction<Number>("_subassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_subassign));

    _script_engine->registerMemberFunction<Number>("_subassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_subassign));
    //...
}
```

Finally, the following code it shows an example of a script that substracts and assigns values from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number-=20 => {0}\",number-=20)\n"
        "Console::outln(\"number-=new Number(30) => {0}\",number-=new Number(30))\n"
    );
    return 0;
}
```

Console output:

```
number-=20 => 0.00
number-=new Number(30) => -30.00
```

_tostring()

Returns custom string when string operation operation is invoved

Syntax

```
String RegisteredType_tostring(ScriptEngine * _script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

A String as a result when string operation operation is invoved

Example

The following code defines a function that converts and returns current value as string,

```
String NumberZs_tostring(ScriptEngine *_script_engine,Number *_this){
    char output[100];
    sprintf(output,"%0.2f",_this->value);
    return output;
}
```

Next, the function *NumberZs_tostring* is registered as member metamethod *tostring_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_tostring",&NumberZs_tostring));

    //...
}
```

Finally, the following code it shows an example of a script that creates a type *Number* and prints its content to the console through *Console::outln*. Because *Console::outln* prints string information it calls *_tostring* implicitly.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"Result _tostring => \"+new Number(10))"
    );

    return 0;
}
```

Console output:

```
Result _tostring => 10.00
```

_xorassign()

Implements *bitwise XOR assignment* operator (aka ^=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredType_xorassign(ScriptEngine *_script_engine, RegisteredType *_this,ParamType *_value);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines two functions that performs a bitwise XOR and assignment to the current instance from a *Float* value or *Number* type respectively,

```
void NumberZs_xorassign(ScriptEngine *_script_engine, Number *_this,zs_float *_n){
    _this->value-=*_n;
}

void NumberZs_xorassign(ScriptEngine *_script_engine, Number *_this,Number *_n){
    _this->value-=_n->value;
}
```

Next, each function *NumberZs_xorassign* is registered as member metamethod *xorassign_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerMemberFunction<Number>("_xorassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_xorassign));

    _script_engine->registerMemberFunction<Number>("_xorassign",static_cast<void (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_xorassign));
    //...

}
```

Finally, the following code it shows an example of a script that perform bitwise XOR and assignment from a *Float* value and *Number* object.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(0);\n"
        "Console::outln(\"number^=0xa => {0}\",number^=0xa);\n"
        "Console::outln(\"number^=new Number(0x9) => {0}\",number^=new Number(0x9))\n"
    );

    return 0;
}
```

Console output:

```
number^=0xa => 10.00
number^=new Number(0x9) => 3.00
```

Static metamethods

_add()

Implements *add* operator (aka +) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_add(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of add operation

Example

The following code defines three functions that performs *add* operation from *Float* value or *Number* type,

```
ClassScriptObject * NumberZs_add(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(_n1->value +_n2->value));
}

ClassScriptObject * NumberZs_add(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number(_n1->value + *_n2));
}

ClassScriptObject * NumberZs_add(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(*_n1 + _n2->value));
}
```

Next, each function *NumberZs_add* is registered as member metamethod *_add* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...

    _script_engine->registerStaticMemberFunction<Number>("_add",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_add));
    _script_engine->registerStaticMemberFunction<Number>("_add",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float *)>(&NumberZs_add));
    _script_engine->registerStaticMemberFunction<Number>("_add",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_add));

    //...
}
```

Finally, the following code it shows an example of a script that performs *_add* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(10) + new Number(20) => \" +  (new Number(10) + new Number(20)));\n"
        "Console::outln(\"new Number(10) + 20 => \" + (new Number(10) + 20));\n"
        "Console::outln(\"10 + new Number(20) => \" + (10 + new Number(20)));\n"
    );

    return 0;
}
```

Console output:

```
new Number(10) + new Number(20) => 30.00
new Number(10) + 20 => 30.00
```

```
10 + new Number(20) => 30.00
```

_and()

Implements *bitwise AND* operator (aka &) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_and(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise AND operation

Example

The following code defines three functions that performs *AND* operation from *Float* value or *Number* type,

```
ClassScriptObject * NumberZs_and(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value & (zs_int)_n2->value));
}

ClassScriptObject * NumberZs_and(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value & (zs_int)*_n2));
}

ClassScriptObject * NumberZs_and(ScriptEngine *_script_engine,zs_float *_n1, Number * _n2){
    return _script_engine->newClassObject(new Number((zs_int)*_n1 & (zs_int)_n2->value));
}
```

Next, each function *NumberZs_and* is registered as member metamethod *_AND* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerStaticMemberFunction<Number>("_and",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_and));
    _script_engine->registerStaticMemberFunction<Number>("_and",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float *)>(&NumberZs_and));
    _script_engine->registerStaticMemberFunction<Number>("_and",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_and));
    //...
}
```

Finally, the following code it shows an example of a script that performs *_AND* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(0x7) & new Number(0x4) => \" +  (new Number(0x7) & new Number(0x04)));\n"
        "Console::outln(\"new Number(0x7) & 0x4 => \" + (new Number(0x7) & 0x04));\n"
        "Console::outln(\"0x7 & new Number(0x4) => \" + (0x7 & new Number(0x04)));\n"
    );

    return 0;
}
```

Console output:

```
new Number(0x7) & new Number(0x4) => 4.00
new Number(0x7) & 0x4 => 4.00
0x7 & new Number(0x4) => 4.00
```

_div()

Implements *division* operator (aka /) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_div(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of division operation

Example

The following code defines three functions that performs *div* operation from *Float* value or *Number* type,

```
ClassScriptObject * NumberZs_div(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(_n1->value / _n2->value));
}

ClassScriptObject * NumberZs_div(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number(_n1->value / *_n2));
}

ClassScriptObject * NumberZs_div(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(*_n1 / _n2->value));
}
```

Next, each function *NumberZs_div* is registered as member metamethod *div_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerStaticMemberFunction<Number>("_div",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_div));
    _script_engine->registerStaticMemberFunction<Number>("_div",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float *)>(&NumberZs_div));
    _script_engine->registerStaticMemberFunction<Number>("_div",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_div));
    //...
}
```

Finally, the following code it shows an example of a script that performs *_div* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(10) / new Number(20) => \" +  (new Number(10) / new Number(20)));\n"
        "Console::outln(\"new Number(10) / 20 => \" + (new Number(10) / 20));\n"
        "Console::outln(\"10 / new Number(20) => \" + (10 / new Number(20)));\n"
    );

    return 0;
}
```

Console output:

```
new Number(10) / new Number(20) => 0.50
new Number(10) / 20 => 0.50
10 / new Number(20) => 0.50
```

_equ()

Implements *equal* operator (aka ==) between first operand and second operand

Syntax

```
bool RegisteredType_equ(RegisteredType *_op1, RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 and op2 are EQUAL
- False if op1 and op2 are NOT EQUAL

Example

The following code defines three functions that performs *equ* operation from *Float* value or *Number* type,

```
bool  NumberZs_equ(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _n1->value ==_n2->value;
}

bool  NumberZs_equ(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _n1->value == *_n2;
}

bool NumberZs_equ(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return *_n1 == _n2->value;
}
```

Next, each function *NumberZs_equ* is registered as member metamethod *_equ* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerStaticMemberFunction<Number>("_equ",static_cast<bool (*)(ScriptEngine *_script_engine,zs_float *,Number * )>(
&NumberZs_equ));
    _script_engine->registerStaticMemberFunction<Number>("_equ",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_equ));
    _script_engine->registerStaticMemberFunction<Number>("_equ",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_equ));
    //...
}
```

Finally, the following code it shows an example of a script that performs *equ* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(20) == new Number(20) => \"+(new Number(20) == new Number(20)))\n"
        "Console::outln(\"new Number(20) == new Number(30) => \"+(new Number(20) == new Number(30)))\n"
    );
    return 0;
}
```

Console output:

```
new Number(20) == new Number(20) => true
new Number(20) == new Number(30) => false
```

_gt()

Implements *greater than* operator (aka >) between first operand and second operand

Syntax

```
bool RegisteredType_gt(RegisteredType *_op1, RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 is GREATHER THAN op2
- False if op1 is LESS OR EQUAL THAN op2

Example

The following code defines three functions that performs *gt* operation from *Float* value or *Number* type,

```
bool  NumberZs_gt(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _n1->value >_n2->value;
}

bool  NumberZs_gt(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _n1->value > *_n2;
}

bool NumberZs_gt(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return *_n1 > _n2->value;
}
```

Next, each function *NumberZs_gt* is registered as member metamethod *gt* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerStaticMemberFunction<Number>("_gt",static_cast<bool (*)(ScriptEngine *_script_engine,zs_float *,Number * )>(
&NumberZs_gt));
    _script_engine->registerStaticMemberFunction<Number>("_gt",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_gt));
    _script_engine->registerStaticMemberFunction<Number>("_gt",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_gt));

    //...

}
```

Finally, the following code it shows an example of a script that performs *gt* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(20) > new Number(20) => \"+(new Number(20) > new Number(20)))\n"
        "Console::outln(\"new Number(40) > new Number(30) => \"+(new Number(40) > new Number(30)))\n"
    );

    return 0;
}
```

Console output:

```
new Number(20) > new Number(20) => false
```

```
new Number(40) > new Number(30) => true
```

_gte()

Implements *greater than or equal* operator (aka >=) between first operand and second operand

Syntax

```
bool RegisteredType_gte(RegisteredType *_op1, RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 is GREATHER THAN OR EQUAL op2
- False if op1 is LESS THAN op2

Example

The following code defines three functions that performs *equ* operation from *Float* value or *Number* type,

```
bool  NumberZs_gte(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _n1->value >=_n2->value;
}

bool  NumberZs_gte(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _n1->value >= *_n2;
}

bool NumberZs_gte(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return *_n1 >= _n2->value;
}
```

Next, each function *NumberZs_gte* is registered as member metamethod *_gte* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...
    _script_engine->registerMemberConstructor<Number>(static_cast<void (*)(zetscript::ScriptEngine *_script_engine,Number *_this,
zetscript::zs_float *_value)>(&NumberZs_set));

    _script_engine->registerMemberConstructor<Number>(static_cast<void (*)(zetscript::ScriptEngine *_script_engine,Number *_this, Number
*_value)>(&NumberZs_set));
    //...

}
```

Finally, the following code it shows an example of a script that performs *gte* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(10) >= new Number(20) => \"+(new Number(10) >= new Number(20)))\n"
        "Console::outln(\"new Number(20) >= new Number(20) => \"+(new Number(20) >= new Number(20)))\n"
        "Console::outln(\"new Number(30) >= new Number(20) => \"+(new Number(30) >= new Number(20)))\n"
    );

    return 0;
}
```

Console output:

```
new Number(10) >= new Number(20) => false
new Number(20) >= new Number(20) => true
new Number(30) >= new Number(20) => true
```

_lt()

Implements *less than* operator (aka <) between first operand and second operand

Syntax

```
bool RegisteredType_lt(RegisteredType *_op1, RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 is LESS THAN op2
- False if op1 is GRATHER EQUAL THAN op2

Example

The following code defines three functions that performs *lt* operation from *Float* value or *Number* type,

```
bool  NumberZs_lt(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _n1->value <_n2->value;
}

bool  NumberZs_lt(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _n1->value < *_n2;
}

bool NumberZs_lt(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return *_n1 < _n2->value;
}
```

Next, each function *NumberZs_lt* is registered as member metamethod *lt* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerStaticMemberFunction<Number>("_lt",static_cast<bool (*)(ScriptEngine *_script_engine,zs_float *,Number * )>(
&NumberZs_lt));
    _script_engine->registerStaticMemberFunction<Number>("_lt",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_lt));
    _script_engine->registerStaticMemberFunction<Number>("_lt",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_lt));
    //...
}
```

Finally, the following code it shows an example of a script that performs *lt* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(20) < new Number(20) => \"+(new Number(20) < new Number(20)))\n"
        "Console::outln(\"new Number(20) < new Number(30) => \"+(new Number(20) < new Number(30)))\n"
    );

    return 0;
}
```

Console output:

```
new Number(20) < new Number(20) => false
new Number(20) < new Number(30) => true
```

_lte()

Implements *less than or equal* operator (aka <=) between first operand and second operand

Syntax

```
_lte(_op1,_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 is LESS THAN OR EQUAL op2
- False if op1 is GRATHER THAN op2

Example

The following code defines three functions that performs *lte* operation from *Float* value or *Number* type,

```
bool  NumberZs_lte(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _n1->value <=_n2->value;
}

bool  NumberZs_lte(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _n1->value <= *_n2;
}

bool NumberZs_lte(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return *_n1 <= _n2->value;
}
```

Next, each function *NumberZs_lte* is registered as member metamethod *lte_ through* ScriptEngine::registerMemberFunction__,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerStaticMemberFunction<Number>("_lte",static_cast<bool (*)(ScriptEngine *_script_engine,zs_float *,Number * )>(
&NumberZs_lte));
    _script_engine->registerStaticMemberFunction<Number>("_lte",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_lte));
    _script_engine->registerStaticMemberFunction<Number>("_lte",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_lte));
    //...
}
```

Finally, the following code it shows an example of a script that performs *lte* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(10) <= new Number(20) => \"+(new Number(10) <= new Number(20)))\n"
        "Console::outln(\"new Number(20) <= new Number(20) => \"+(new Number(20) <= new Number(20)))\n"
        "Console::outln(\"new Number(30) <= new Number(20) => \"+(new Number(30) <= new Number(20)))\n"
    );

    return 0;
}
```

Console output:

```
new Number(10) <= new Number(20) => true
new Number(20) <= new Number(20) => true
new Number(30) <= new Number(20) => false
```

_mod()

Implements *modulus* operator (aka %) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_mod(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of modulus operation

Example

The following code defines three functions that performs *mod* operation from *Float* value or *Number* type,

```
ClassScriptObject * NumberZs_mod(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(fmod(_n1->value,_n2->value)));
}

ClassScriptObject * NumberZs_mod(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number(fmod(_n1->value, *_n2)));
}

ClassScriptObject * NumberZs_mod(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(fmod(*_n1, _n2->value)));
}
```

Next, each function *NumberZs_mod* is registered as member metamethod *_mod* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerStaticMemberFunction<Number>("_mod",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_mod));
    _script_engine->registerStaticMemberFunction<Number>("_mod",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float *)>(&NumberZs_mod));
    _script_engine->registerStaticMemberFunction<Number>("_mod",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_mod));
    //...
}
```

Finally, the following code it shows an example of a script that performs *_mod* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(10) % new Number(20) => \" +  (new Number(10) % new Number(20)));\n"
        "Console::outln(\"new Number(10) % 20 => \" + (new Number(10) % 20));\n"
        "Console::outln(\"10 % new Number(20) => \" + (10 % new Number(20)));\n"
    );
    return 0;
}
```

Console output:

```
new Number(10) % new Number(20) => 10.00
new Number(10) % 20 => 10.00
10 % new Number(20) => 10.00
```

_mul()

Implements *multiplication* operator (aka *) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_mul(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of multiplication operation

Example

The following code defines three functions that performs *mul* operation from *Float* value or *Number* type,

```
ClassScriptObject * NumberZs_mul(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(_n1->value * _n2->value));
}

ClassScriptObject * NumberZs_mul(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number(_n1->value * (*_n2)));
}

ClassScriptObject * NumberZs_mul(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(*_n1 * _n2->value));
}
```

Next, each function *NumberZs_mul* is registered as member metamethod *_mul* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerStaticMemberFunction<Number>("_mul",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_mul));
    _script_engine->registerStaticMemberFunction<Number>("_mul",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float *)>(&NumberZs_mul));
    _script_engine->registerStaticMemberFunction<Number>("_mul",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_mul));
    //...
}
```

Finally, the following code it shows an example of a script that performs *mul* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(10) * new Number(20) => \" +  (new Number(10) * new Number(20)));\n"
        "Console::outln(\"new Number(10) * 20 => \" + (new Number(10) * 20));\n"
        "Console::outln(\"10 * new Number(20) => \" + (10 * new Number(20)));\n"
    );

    return 0;
}
```

Console output:

```
new Number(10) * new Number(20) => 200.00
new Number(10) * 20 => 200.00
10 * new Number(20) => 200.00
```

_nequ()

Implements *not equal* operator (aka !=) between first operand and second operand

Syntax

```
bool RegisteredType_nequ(RegisteredType *_op1, RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- True if op1 and op2 are NOT EQUAL
- False if op1 and op2 are EQUAL

Example

The following code defines three functions that performs *equ* operation from *Float* value or *Number* type,

```
bool  NumberZs_nequ(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _n1->value !=_n2->value;
}

bool  NumberZs_nequ(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _n1->value != *_n2;
}

bool NumberZs_nequ(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return *_n1 != _n2->value;
}
```

Next, each function *NumberZs_nequ* is registered as member metamethod *_nequ* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerStaticMemberFunction<Number>("_nequ",static_cast<bool (*)(ScriptEngine *_script_engine,zs_float *,Number * )>
(&NumberZs_nequ));
    _script_engine->registerStaticMemberFunction<Number>("_nequ",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,zs_float *)>(
&NumberZs_nequ));
    _script_engine->registerStaticMemberFunction<Number>("_nequ",static_cast<bool (*)(ScriptEngine *_script_engine,Number *,Number * )>(
&NumberZs_nequ));

    //...

}
```

Finally, the following code it shows an example of a script that performs *nequ* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(20) != new Number(20) => \"+(new Number(20) != new Number(20)))\n"
        "Console::outln(\"new Number(20) != new Number(30) => \"+(new Number(20) != new Number(30)))\n"
    );

    return 0;
}
```

Console output:

```
new Number(20) != new Number(20) => false
```

```
new Number(20) != new Number(30) => true
```

_or()

Implements *bitwise OR* operator (aka |) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_or(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise OR operation

Example

The following code defines three functions that performs *OR* operation from *Float* value or *Number* type,

```
ClassScriptObject * NumberZs_or(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value | (zs_int)_n2->value));
}

ClassScriptObject * NumberZs_or(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value | (zs_int)*_n2));
}

ClassScriptObject * NumberZs_or(ScriptEngine *_script_engine,zs_float *_n1, Number * _n2){
    return _script_engine->newClassObject(new Number((zs_int)*_n1 | (zs_int)_n2->value));
}
```

Next, each function *NumberZs_or* is registered as member metamethod *_OR* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerStaticMemberFunction<Number>("_or",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_or));
    _script_engine->registerStaticMemberFunction<Number>("_or",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float *)>(&NumberZs_or));
    _script_engine->registerStaticMemberFunction<Number>("_or",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_or));
    //...
}
```

Finally, the following code it shows an example of a script that performs *_OR* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(0x1) | new Number(0x2) => \" +  (new Number(0x1) | new Number(0x2)));\n"
        "Console::outln(\"new Number(0x1) | 0x2 => \" + (new Number(0x1) | 0x2));\n"
        "Console::outln(\"0x1 | new Number(0x2) => \" + (0x1 | new Number(0x2)));\n"
    );

    return 0;
}
```

Console output:

```
new Number(0x1) | new Number(0x2) => 3.00
new Number(0x1) | 0x2 => 3.00
0x1 | new Number(0x2) => 3.00
```

_shl()

Implements *bitwise shift left* operator (aka <<) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_shl(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise SHIFT LEFT operation

Example

```
//...

ClassScriptObject * NumberZs_shl(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value << (zs_int)_n2->value));
}

ClassScriptObject * NumberZs_shl(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value << (zs_int)*_n2));
}

ClassScriptObject * NumberZs_shl(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number((zs_int)*_n1 << (zs_int)_n2->value));
}
```

Next, each function *NumberZs_shl* is registered as member metamethod *_shl* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerStaticMemberFunction<Number>("_shl",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_shl));
    _script_engine->registerStaticMemberFunction<Number>("_shl",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float *)>(&NumberZs_shl));
    _script_engine->registerStaticMemberFunction<Number>("_shl",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_shl));
    //...
}
```

Finally, the following code it shows an example of a script that performs *_shl* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(0x1) << new Number(2) => \" +  (new Number(0x1) << new Number(2)));\n"
        "Console::outln(\"new Number(0x1) << 2 => \" + (new Number(0x1) << 2));\n"
        "Console::outln(\"0x1 << new Number(2) => \" + (0x1 << new Number(2)));\n"
    );

    return 0;
}
```

Console output:

```
new Number(0x1) << new Number(2) => 4.00
new Number(0x1) << 2 => 4.00
0x1 << new Number(2) => 4.00
```

_shr()

Implements *bitwise SHIFT RIGHT* operator (aka <<) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_shr(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise SHIFT RIGHT operation

Example

The following code defines three functions that performs *shr* operation from *Float* value or *Number* type,

```
ClassScriptObject * NumberZs_shr(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value >> (zs_int)_n2->value));
}

ClassScriptObject * NumberZs_shr(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value >> (zs_int)*_n2));
}

ClassScriptObject * NumberZs_shr(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number((zs_int)*_n1 >> (zs_int)_n2->value));
}
```

Next, each function *NumberZs_shr* is registered as member metamethod *_shr* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerStaticMemberFunction<Number>("_shr",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_shr));
    _script_engine->registerStaticMemberFunction<Number>("_shr",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float * )>(&NumberZs_shr));
    _script_engine->registerStaticMemberFunction<Number>("_shr",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_shr));
    //...
}
```

Finally, the following code it shows an example of a script that performs *_shr* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(0x8) >> new Number(2) => \" +  (new Number(0x8) >> new Number(2)));\n"
        "Console::outln(\"new Number(0x8) >> 2 => \" + (new Number(0x8) >> 2));\n"
        "Console::outln(\"0x8 >> new Number(2) => \" + (0x8 >> new Number(2)));\n"
    );

    return 0;
}
```

Console output:

```
new Number(0x8) >> new Number(2) => 2.00
new Number(0x8) >> 2 => 2.00
0x8 >> new Number(2) => 2.00
```

_sub()

Implements *subtraction* operator (aka -) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_sub(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of add operation

Example

The following code defines three functions that performs *sub* operation from *Float* value or *Number* type,

```
ClassScriptObject * NumberZs_sub(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(_n1->value +_n2->value));
}

ClassScriptObject * NumberZs_sub(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number(_n1->value + *_n2));
}

ClassScriptObject * NumberZs_sub(ScriptEngine *_script_engine,zs_float *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number(*_n1 + _n2->value));
}
```

Next, each function *Num_berZs_sub* is registered as member metamethod *_sub* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...

    _script_engine->registerStaticMemberFunction<Number>("_sub",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_sub));
    _script_engine->registerStaticMemberFunction<Number>("_sub",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float *)>(&NumberZs_sub));
    _script_engine->registerStaticMemberFunction<Number>("_sub",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_sub));

    //...
}
```

Finally, the following code it shows an example of a script that performs *_sub* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(10) - new Number(20) => \" +  (new Number(10) - new Number(20)));\n"
        "Console::outln(\"new Number(10) - 20 => \" + (new Number(10) - 20));\n"
        "Console::outln(\"10 - new Number(20) => \" + (10 - new Number(20)));\n"
    );

    return 0;
}
```

Console output:

```
new Number(10) - new Number(20) => -10.00
new Number(10) - 20 => -10.00
10 - new Number(20) => -10.00
```

_xor()

Implements *bitwise XOR* operator (aka ^) between first operand and second operand

Syntax

```
ClassScriptObject * RegisteredType_xor(RegisteredType *_op1,RegisteredType *_op2)
```

Parameters

- *_op1* : 1st operand.
- *_op2* : 2nd operand.

Returns

- A new object as a result of bitwise XOR operation

Example

The following code defines three functions that performs *XOR* operation from *Float* value or *Number* type,

```
ClassScriptObject * NumberZs_xor(ScriptEngine *_script_engine,Number *_n1, Number *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value ^ (zs_int)_n2->value));
}

ClassScriptObject * NumberZs_xor(ScriptEngine *_script_engine,Number *_n1, zs_float *_n2){
    return _script_engine->newClassObject(new Number((zs_int)_n1->value ^ (zs_int)*_n2));
}

ClassScriptObject * NumberZs_xor(ScriptEngine *_script_engine,zs_float *_n1, Number * _n2){
    return _script_engine->newClassObject(new Number((zs_int)*_n1 ^ (zs_int)_n2->value));
}
```

Next, each function *NumberZs_xor* is registered as member metamethod *_XOR* through *ScriptEngine::registerMemberFunction*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){
    //...
    _script_engine->registerStaticMemberFunction<Number>("_xor",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,zs_float
*,Number * )>(&NumberZs_xor));
    _script_engine->registerStaticMemberFunction<Number>("_xor",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,zs_float *)>(&NumberZs_xor));
    _script_engine->registerStaticMemberFunction<Number>("_xor",static_cast<ClassScriptObject * (*)(ScriptEngine *_script_engine,Number *
,Number * )>(&NumberZs_xor));
    //...
}
```

Finally, the following code it shows an example of a script that performs *_XOR* operation from a *Float* and *Number*.

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "Console::outln(\"new Number(0xa) ^ new Number(0x9) => \" +  (new Number(0xa) ^ new Number(0x9)));\n"
        "Console::outln(\"new Number(0xa) ^ 0x9 => \" + (new Number(0xa) ^ 0x9));\n"
        "Console::outln(\"0xa ^ new Number(0x9) => \" + (0xa ^ new Number(0x9)));\n"
    );

    return 0;
}
```

Console output:

```
new Number(0xa) ^ new Number(0x9) => 3.00
new Number(0xa) ^ 0x9 => 3.00
0xa ^ new Number(0x9) => 3.00
```

3.4.5. Properties

ZetScript API allows to register class properties as we could see in Language Class Properties.

Register constant property

Returns a constant property.

Syntax

```
ReturnType RegisteredTypeProperty_get(ScriptEngine *_script_engine, RegisteredType *_this)
```

Parameters

- *_script_engine* : ScriptEngine instance.
- *_this* : The current instance.

Returns

Returns the value of the property

Example

The following code defines a function that returns the *value* contents of current instance,

```
zs_float NumberZs_MAX_VALUE_get(ScriptEngine *_script_engine,Number *_this){
    return FLT_MAX;
}
```

Next, the function *NumberZs_MAX_VALUE_get* is registered through *ScriptEngine::registerConstMemberProperty*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerConstMemberProperty<Number>("MAX_VALUE",NumberZs_MAX_VALUE_get);

    //...

}
```

Finally, the following code it shows an example of a script that returns *Number::MAX_VALUE* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number();\n"
        "Console::outln(\"number::MAX_VALUE => {0}\",number::MAX_VALUE)\n"
    );

    return 0;
}
```

Console output:

```
number::MAX_VALUE => 0.00
```

Register property metamethods

_addassign()

Implements *addition assignment* operator (aka +=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_addassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs and *addassign* to *value* variable of current instance,

```
void NumberZs_value_addassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value+=*_value;
}
```

Next, the function *NumberZs_value_addassign* is registered as member metamethod *addassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_addassign",NumberZs_value_addassign));

    //...

}
```

Finally, the following code it shows an example of a script that performs a *addassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number.value+=10 => {0}\",number.value+=10)\n"
    );

    return 0;
}
```

Console output:

```
number.value+=10 => 30.000000
```

_andassign()

Implements *bitwise AND assignment* operator (aka &=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_andassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs and *andassign* to *value* variable of current instance,

```
void NumberZs_value_andassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value=(int)(_this->value)&(int)(*_value);
}
```

Next, the function *NumberZs_value_andassign* is registered as member metamethod *andassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_andassign",NumberZs_value_andassign);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *andassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(0xf);\n"
        "Console::outln(\"number.value&=0x7 => {0}\",number.value&=0x7)\n"
    );

    return 0;
}
```

Console output:

```
number.value&=0x7 => 7.000000
```

_divassign()

Implements *division assignment* operator (aka /=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_divassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs and *divassign* to *value* variable of current instance,

```
void NumberZs_value_divassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value/=*_value;
}
```

Next, the function *NumberZs_value_divassign* is registered as member metamethod *divassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_divassign",NumberZs_value_divassign);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *divassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number.value/=20 => {0}\",number.value/=20)\n"
    );

    return 0;
}
```

Console output:

```
number.value/=20 => 1.000000
```

_get()

_get returns the value of the property

Syntax

```
ReturnType RegisteredTypeProperty_get(ScriptEngine *_script_engine, RegisteredType *_this)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

Returns the value of the property

Example

The following code defines a function as a *get* property operation that returns the *value* contents of current instance,

```
zs_float NumberZs_value_get(ScriptEngine *_script_engine,Number *_this){
    return _this->value;
}
```

Next, the function *NumberZs_value_get* is registered as member metamethod *get* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_get",NumberZs_value_get);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *get* operation from *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number.value => {0}\",number.value)\n"
    );

    return 0;
}
```

Console output:

```
number.value => 20.000000
```

_modassign()

Implements *modulus assignment* operator (aka %=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_modassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs and *modassign* to *value* variable of current instance,

```
void NumberZs_value_modassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value=fmod(_this->value,*_value);
}
```

Next, the function *NumberZs_value_modassign* is registered as member metamethod *addassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_modassign",NumberZs_value_modassign);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *modassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(250);\n"
        "Console::outln(\"number.value%=30 => {0}\",number.value%=30)\n"
    );

    return 0;
}
```

Console output:

```
number.value%=30 => 10.000000
```

_mulassign()

Implements *multiplication assignment* operator (aka *=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_mulassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs and *mulassign* to *value* variable of current instance,

```
void NumberZs_value_mulassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value*=*_value;
}
```

Next, the function *NumberZs_value_mulassign* is registered as member metamethod *mulassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_mulassign",NumberZs_value_mulassign);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *mulassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number.value*=2 => {0}\",number.value*=2)\n"
    );

    return 0;
}
```

Console output:

```
number.value*=2 => 40.000000
```

_orassign()

Implements *bitwise OR assignment* operator (aka |=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_orassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs and *orassign* to *value* variable of current instance,

```
void NumberZs_value_orassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value=(int)(_this->value)|(int)(*_value);
}
```

Next, the function *NumberZs_value_orassign* is registered as member metamethod *orassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_orassign",NumberZs_value_orassign);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *orassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(0x1);\n"
        "Console::outln(\"number.value|=0x2 => {0}\",number.value|=0x2)\n"
    );

    return 0;
}
```

Console output:

```
number.value|=0x2 => 3.000000
```

_postdec()

Implements *post_decrement* operator (aka a--)

Syntax

```
ReturnType  RegisteredType_value_postdec(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

Return the value before perform post decrement operation. The return value can be one defined in return types.

Example

The following code defines a function that performs a *postdec* of current instance,

```
zs_float  NumberZs_value_postdec(ScriptEngine *_script_engine,Number *_this){
    return _this->value--;
}
```

Next, the function *NumberZs_value_postdec* is registered as member metamethod *postdec* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_postdec",NumberZs_postdec);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *postdec* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number.value-- => {0}\",number.value--)\n"
    );

    return 0;
}
```

Console output:

```
number.value-- => 20.000000
```

_postinc()

Implements *post_increment* operator (aka a++)

Syntax

```
ReturnType  RegisteredType_value_postinc(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

Return the value before perform post increment operation. The return value can be one defined in return types.

Example

The following code defines a function that performs a *postinc* of current instance,

```
zs_float  NumberZs_value_postinc(ScriptEngine *_script_engine,Number *_this){
    return _this->value++;
}
```

Next, the function *NumberZs_value_postinc* is registered as member metamethod *postinc* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberFunction<Number>("_postinc",NumberZs_postinc);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *postinc* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"number.value++ => {0}\",number.value++)\n"
    );

    return 0;
}
```

Console output:

```
number.value++ => 20.000000
```

_predec()

```
ReturnType  RegisteredType_value_predec(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

Return the value before perform pre decrement operation. The return value can be one defined in return types.

Example

The following code defines a function that performs a *predec* of current instance,

```
zs_float  NumberZs_value_predec(ScriptEngine *_script_engine,Number *_this){
    return --_this->value;
}
```

Next, the function *NumberZs_value_predec* is registered as member metamethod *predec* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_predec",NumberZs_value_predec);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *predec* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"--number.value => {0}\",--number.value)\n"
    );

    return 0;
}
```

Console output:

```
--number.value => 19.000000
```

_preinc()

Implements *pre_increment* operator (aka ++a)

Syntax

```
ReturnType  RegisteredType_value_preinc(ScriptEngine *_script_engine, RegisteredType *_this);
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance

Returns

Return the value before perform pre increment operation. The return value can be one defined in return types.

Example

The following code defines a function that performs a *preinc* of current instance,

```
zs_float  NumberZs_value_preinc(ScriptEngine *_script_engine,Number *_this){
    return ++_this->value;
}
```

Next, the function *NumberZs_value_preinc* is registered as member metamethod *preinc* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_preinc",NumberZs_value_preinc);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *preinc* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"++number.value => {0}\",++number.value)\n"
    );

    return 0;
}
```

Console output:

```
++number.value => 21.000000
```

_set()

Implements *assignment* operator (aka =) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_set(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs a *set* to *value* variable of current instance,

```
void NumberZs_value_set(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value=*_value;
}
```

Next, the function *NumberZs_value_set* is registered as member metamethod *set* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_set",NumberZs_value_set);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *set* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(20);\n"
        "Console::outln(\"On operation 'number=new Number(20)' then 'number.value' is => \"+number.value)\n"
        "number.value = 10;\n"
        "Console::outln(\"On operation 'number=10' the 'number.value' is => \"+number.value)\n"
    );

    return 0;
}
```

Console output:

```
On operation 'number=new Number(20)' then 'number.value' is => 20.000000
On operation 'number=10' the 'number.value' is => 10.000000
```

_shlassign()

Implements *bitwise SHIFT LEFT assignment* operator (aka <<=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_shlassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs and *shlassign* to *value* variable of current instance,

```
void NumberZs_value_shlassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value=(int)(_this->value)<<(int)(*_value);
}
```

Next, the function *NumberZs_value_shlassign* is registered as member metamethod *shlassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_shlassign",NumberZs_value_shlassign);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *shlassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(0x1);\n"
        "Console::outln(\"number.value<<=1 => {0}\",number.value<<=1)\n"
    );

    return 0;
}
```

Console output:

```
number.value<<=1 => 2.000000
```

_shrassign()

Implements *bitwise SHIFT RIGHT assignment* operator (aka >>=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_shrassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs and *shrassign* to *value* variable of current instance,

```
void NumberZs_value_shrassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value=(int)(_this->value)>>(int)(*_value);
}
```

Next, the function *NumberZs_value_shrassign* is registered as member metamethod *shrassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_shrassign",NumberZs_value_shrassign);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *shrassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number = new Number(0x8);\n"
        "Console::outln(\"number.value>>=1 => {0}\",number.value>>=1)\n"
    );

    return 0;
}
```

Console output:

```
number.value>>=1 => 4.000000
```

_subassign()

Implements *substraction assignment* operator (aka -=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_subassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs a *subassign* to *value* variable of current instance,

```
void NumberZs_value_subassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value-=*_value;
}
```

Next, the function *NumberZs_value_subassign* is registered as member metamethod *subassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_subassign",NumberZs_value_subassign);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *subassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(30);\n"
        "Console::outln(\"number.value-=20 => {0}\",number.value-=20)\n"
    );

    return 0;
}
```

Console output:

```
number.value-=20 => 10.000000
```

_xorassign()

Implements *bitwise XOR assignment* operator (aka ^=) with a value entered by parameter as right operand.

Syntax

```
void RegisteredTypeProperty_xorassign(ScriptEngine *_script_engine, RegisteredType *_this, ParamType *_value)
```

Parameters

- *_script_engine* : ScriptEngine instance
- *_this* : The current instance
- *_value* : Value or variable as right operand. Its type can be one defined in parameter types.

Returns

None.

Example

The following code defines a function that performs and *xorassign* to *value* variable of current instance,

```
void NumberZs_value_xorassign(ScriptEngine *_script_engine,Number *_this,zs_float *_value){
    _this->value=(int)(_this->value)^(int)(*_value);
}
```

Next, the function *NumberZs_value_xorassign* is registered as member metamethod *xorassign* through *ScriptEngine::registerMemberPropertyMetamethod*,

```
void NumberZs_register(zetscript::ScriptEngine *_script_engine){

    //...

    _script_engine->registerMemberPropertyMetamethod<Number>("value","_xorassign",NumberZs_value_xorassign);

    //...

}
```

Finally, the following code it shows an example of a script that performs a *xorassign* operation to *Number::value* property,

```
#include "NumberZs.h"

int main(){

    zetscript::ScriptEngine script_engine;

    NumberZs_register(&script_engine);

    script_engine.compileAndRun(
        "var number=new Number(0x0);\n"
        "Console::outln(\"number.value^=0xa => {0}\",number.value^=0xa)\n"
    );

    return 0;
}
```

Console output:

```
number.value^=0xa => 10.000000
```